

DETERMINISTIC MULTIPROCESSOR SCHEDULING
FOR MIMD COMPUTER SYSTEMS

By

FRANK D. ANGER

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN
PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1987

ACKNOWLEDGMENTS

The opportunity to give thanks to people and institutions which have taken part in the creation of this dissertation is one which I heartily embrace; reaching this point, however, has cut so deeply into the fabric of my life and the lives of my family, that weighing the effects of the laughter of my children and the advice of my professors becomes a difficult enterprise. Nonetheless, I would like first to specifically thank those who consciously acted as counselor or gave assistance in this work. My dissertation advisor, Dr. Yuan-Chieh Chow, has provided the environment, guidance, and many ideas which made the research possible. Each of the other dissertation committee members also contributed to the work: Dr. Louis Martin-Vega has been a frequent source of information, perspective, and enthusiasm; Dr. Douglas Dankel has provided constant support, detailed editorial criticism, and perennial good judgement; Dr. Stephen Thebaut has often brought sound advice and, just as often, good humor; and Dr. Gerhard Ritter has cooperated throughout the research.

Beyond the committee, I am grateful to have had the opportunity to work with three undergraduates whose senior projects have made direct and substantial contribution to the research presented here: Dennis Suppe, Borden Wilson, and Michael Ellis did fine jobs of developing different aspects of the scheduling simulator and carrying out arduous portions of the data collection and analysis. Moreover, Dennis's energy, optimism, and conversation kept all four of us going.

There is, however, one person to whom my debt is indeed great. Dr. Jing-Jang Hwang has provided ideas, critique, vision, and hard work. He has also been the other half of many long discussions which have given substance to the research and

larger difficulties; and, finally, he is responsible for putting this work into that mysterious form that makes it print out so beautifully.

On the other hand, there are many people who have contributed to this research perhaps unknowingly. Dr. Carlos Segami set the example and gave the first impetus toward changing roles from professor to student, while the University of Puerto Rico gave its moral and monetary support to my adventure. From that institution, Professor Brunilda Nazario and Oliva Loperena, in particular, assisted greatly to make our time in Gainesville more trouble free. Dr. Roger Elliott, then chairman of the CIS Department at the University of Florida, likewise gave encouragement of many kinds.

My mother, Julia Anger, has patiently watched this venture and, as always, given it her blessing. Whether or not they think they helped at all, I must thank my three sons—Angel, Gus, and Art—for doing what young people do so well, that fills us as parents with awe; and I also thank them for understanding when the computer and books must have seemed more important to me than they. Finally there is one person whose contribution and support were constant. My wife, Rita, gave me the greatest encouragement. Her incredible dynamism and her—as I would often tell her—unfounded confidence kept me going when I would have gladly wavered. As my companion in study, my co-worker, my most tenacious critic, and my source of inspiration she has helped in more ways than can be described in character strings. To her belongs my eternal gratitude.

TABLE OF CONTENTS

	Page
ACKNOWLEDGMENTS	ii
ABSTRACT	vi
CHAPTER	
I BACKGROUND	1
Scheduling	2
Computer Architecture	6
Concurrent Programming.....	8
Performance Analysis	9
II TURNAROUND TIME IN A GENERAL PURPOSE MPS.....	13
Dynamic and Static Scheduling Problems	14
Analytic Approach to Heuristic Algorithms.....	18
Multiprocessor Simulator.....	28
Simulation Results	30
Towards a Theory of Program Size.....	38
Conclusions	43
III LOOSELY COUPLED SYSTEMS	45
Scheduling and Communication.....	46
An Algorithm for Precedence Trees	49
Extensions	57
IV OTHER MULTIPROCESSOR SCHEDULING PROBLEMS.....	68
More MIMD Scheduling Problems	68
SMD and Specialized Architecture Problems	73
Open Questions.....	74
BIBLIOGRAPHY	76
GLOSSARY	81

APPENDICES	83
A RESULTS OF FRIEDMAN ANALYSIS.....	83
B STATISTICAL TEST RESULTS	90
BIOGRAPHICAL SKETCH.....	91

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

DETERMINISTIC MULTIPROCESSOR SCHEDULING
FOR MIMD COMPUTER SYSTEMS

By

FRANK D. ANGER

December 1987

Chairman: Dr. Yuan-Chieh Chow
Major Department: Computer and Information Sciences

The research reported contributes to the theory of scheduling as it applies to modern general-purpose multiple processor systems. Two distinct deterministic models are considered.

With the first model for tightly coupled systems, a study is made of the efficiency of scheduling policies for minimizing the average turnaround time of a set of independent jobs, each consisting of a collection of schedulable subtasks obeying a precedence relation. A number of policies are defined based on the well-known Shortest Job First (SJF) algorithm, and a simulation study is made comparing their performance. Analysis of the results reveals no clear winner. On the other hand, best case and worst case bounds are obtained for one of the algorithms, called CSJF, which indicate, in particular, that it can do no worse than its sequential counterpart, SJF. Moreover, CSJF is shown to be asymptotically optimal if the length of the individual jobs is bounded. Finally, a new measure of the size of a job is proposed as the basis of a new heuristic algorithm for this problem. It is further shown that the size

so defined is more closely related to the optimal makespan (completion time) of the job when run on m processors than either the critical path time or the total processing time.

With the second model, for loosely coupled systems, algorithms are developed to minimize the makespan of a set of precedence related tasks when run on an m -processor system in which communication delays are not negligible. A number of basic assumptions are made on the interconnection architecture and the communication protocol in order to treat the system deterministically. Additionally, the principal results apply under the assumptions Fully Connected and Identical Links. An efficient algorithm, Join Latest Predecessor (JLP), is developed and shown to be optimal in case the precedence relation on the tasks form an "opposing forest" and the system satisfies two additional assumptions: Sufficient Processors and Short Communication Delays. Two polynomial-time extensions of the JLP algorithm--EJLP and JLP/D--are presented: the first is conjectured to be optimal when there are not sufficient processors, while the latter is proved to be optimal for arbitrary precedence relations but with sufficient processors.

CHAPTER I BACKGROUND

From smoky factories and crystal-walled executive suites, from humming computer centers and cluttered principals' offices, from a whole spectrum of sources come the day-to-day problems which engender the begrudgingly given planning steps known as "scheduling." In most of these environments, scheduling begins as orders to "get it done by 4 o'clock" or as the first-come-first-served reflex to a demanding clientele. And this is where it may end. But sometimes, long experience teaches that more serious planning may lead to greater productivity, more free time, smoother operations, or greater profits. Scheduling may have humble beginnings in the tediously repeated tasks at a myriad of similar workstations, or more glamorous ones in the inner sanctums of larger organizations, where long-term projects are born and nurtured. Here scheduling takes on a more respected air, and its necessity and benefits are more clearly recognized. The larger and longer a project, the more essential it becomes that all the pieces fit together correctly, and the harder it becomes for a single person to visualize and coordinate all its components. Thus, a theory of scheduling is born which attempts to study this diverse problem area and produce rules for completing effective planning.

The body of literature which has grown up self-consciously referring to scheduling as a discipline covers half a century and an ever-broadening range of problems. It has produced theoretical and practical solutions to some of these problems, and it has shown that some of them are beyond the abilities of the most modern computers to solve in an optimal fashion. Small twists in the constraints and conditions under which a problem is posed can turn an easy exercise into an amazingly difficult

computation. Many mathematical and computational techniques have been brought to bear on these problems: exhaustive search, queuing theory, linear programming, combinatorial methods, statistics, simulations, and more.

The principal objective of this dissertation is to contribute to the theory supporting the efficient use of multiple processor systems (MPS). Although there are many ways to increase efficiency, this work considers only a few specific methods related to the scheduling of the programs to be executed by an MPS. Even within this apparently narrow area there lie a large number of different problems and methods of solution. In order to describe the research and put it into perspective within the scope of more efficient use of the MPS, it is necessary to discuss both the previous results in scheduling theory and the characteristics of MPSs. The first chapter addresses itself to this effort. Chapter II investigates a class of scheduling problems which are particularly appropriate for a shared-memory multiprocessor system running concurrent programs. Both analytic and simulation methods are used in this chapter. Chapter III follows with some interesting results for scheduling on loosely coupled systems with significant overhead due to interprocessor communications. The last chapter indicates a variety of other related problems and possibilities for future research. Because of the large number of specialized mnemonics in this work, a glossary has been included for convenient reference. Also included are summaries of statistical tests associated with the simulation reported in Chapter II.

1.1. Scheduling

What constitutes a scheduling problem is not always well defined. It can range from simple sequencing of events to a complex decision process affecting the allocation of a variety of resources and the timing of a number of different types of operations. The sole interest of this work is in problems of when and where to execute

program "tasks" within a MPS in order to optimize some performance measure. Some of the problems considered also take into account the communication overhead incurred by sending messages from one task to another; however, the actual scheduling of messages is not considered.

One of the most application-dependent and subjective criteria in scheduling is the objective function: What is to be optimized? The motivation may be profitability, efficiency, user satisfaction, or some other criteria. Some possible examples appear in Table 1.1.

Table 1.1 Some Performance Objectives

APPLICATION	OBJECTIVE
Robot control	Minimize makespan
Data processing	Maximize throughput
Scientific	Maximize throughput
Real time system	Minimize number of late jobs
On-line database	Minimize response time
Interactive multiprog.	Min. average turnaround time

The particular assumptions that will be made on all the scheduling problems are as follows:

1. The system has m identical processors where m is greater than one.
2. At any given moment, the system has a fixed collection of tasks, T_i , to execute, each with a fixed, known processing time, u_i . (This is a "deterministic" scheduling problem.)
3. Once a task is assigned to a processor, the task must be run to completion on that processor without interruption. (This is a "non-preemptive" problem.)
4. Any given processor can only run a single task at a time.
5. If there is a "precedence relation" among the tasks, then no task can be scheduled to run before all its predecessors have run to completion.

6. In some problems, the whole collection of tasks is known at the outset ("static" problem), while in others, tasks arrive at different times and nothing is known of them until they arrive (This is a "dynamic" problem).
7. The performance measure to be optimized will always be either
 - (a) Total time to complete the set of tasks (makespan), or
 - (b) Average turnaround time running a set of independent jobs.

In order to give some perspective to the state of scheduling theory today, it is necessary to discuss these assumptions. Some of them are standard to most scheduling problems, but others restrict the research considerably. The first rules out the large class of single-processor scheduling problems, while the second eliminates the whole area of non-deterministic scheduling problems, which are frequently analyzed through queuing theory and other statistical methods. Additionally, Assumption 3 limits the investigation to the non-preemptive problems, most of which have a corresponding preemptive problem. Although the theory is equally well developed for the areas thereby left out, nothing further will be said about them. On the other hand, Assumption 7 gives a very specific focus to the rest of this dissertation, making it appropriate to talk briefly in this chapter about other possible objective functions.

In many situations, particularly real-time programming, it is required that answers be obtained within a given time limit; otherwise their value degrades or they become worthless. In such cases, minimizing the number of late jobs or minimizing the maximum turnaround time is more appropriate objectives than the ones selected for investigation. On the other hand, a computer center director handling batch data-processing jobs might be most interested in the total volume of work he can finish in a given time, as measured in amount of output, number of completed jobs, or number of seconds of "useful" CPU time. In some applications, a "deadline" is given for each task, and it may be required either to minimize the total "tardiness"

(total time spent after the deadlines running late tasks) or the total "lateness" (the sum of the differences [finish time - deadline], some of which may be negative). Further discussion of possible objectives is found in Section 4 of this chapter.

There are other kinds of scheduling problems which differ from the ones discussed so far because they assume different kinds of processors and different kinds of tasks. In these problems, some tasks must be performed only on certain processors or must be performed on some combination of processors in some given order. Such "job-shop" scheduling problems relate to many industrial and assembly-line environments, but can also be used to model the situation in a computing system when scheduling of the input-output processors is included in the problem. Chapter IV presents more types of scheduling problems.

In 1981, Lageweg, Lawler, Lenstra, and Rinnooy Kan [LAGE81a, LAGE81b] published a computerized classification of results for a very wide variety of deterministic scheduling problems. These problems were presented using a formal scheme for describing and classifying the different kinds of scheduling problems based on three major parameters:

- (a) the number and kinds of processors;
- (b) the job characteristics: preemptive or not, the type of precedence relation; and other restrictions on start times and finish times, and
- (c) the objective function to be optimized.

They used a notation that is similar to the one used in queuing theory to describe the type of problem under discussion: $P/tree/\sum C_j$, for example, represents the problem of an arbitrary number of identical processors (P) and a set of tasks satisfying a tree-shaped precedence relation with the objective of minimizing the sum of the completion times (hence minimizing the average turnaround time). The authors of the scheme further observe a partial ordering on the difficulty of the problems being classified: for example, both minimizing maximum lateness and minimizing average

turnaround time can be accomplished by an algorithm capable of minimizing the total (or average) tardiness. In this way, they are able to present information on the "maximal" easy problems--ones with known polynomial-time solutions such that no more "difficult" problem has been solved--and "minimal" hard problems--ones which are NP-hard and such that no "easier" problem is known to be NP-hard. In 1981, this scheme applied to the literature on scheduling was able to classify 4536 scheduling problems into 416 easy, 432 open, and 3688 hard problems [LAGE81b].

Although most of the "mainstream" scheduling problems can be classified under the foregoing scheme, there is still much literature on deterministic scheduling which does not fall into these divisions. Work on scheduling with a number of additional scarce "resources," scheduling with set-up and tear-down times, and scheduling with variable numbers of processors are examples of further problems which have received attention in the literature. Chapter III of this dissertation gives attention to yet another kind of scheduling problem, in which there are significant time delays associated with scheduling a task and its immediate successor on separate processors. Such problems provide a more apt model of loosely-coupled computer systems than do the traditional models.

1.2. Computer Architecture

Classified scheduling theory considers communication time to be negligible, implying that the actual computer architecture does not enter into the problem. When there are substantial delays due to the communication between processors, however, the method of interconnecting the processors becomes relevant to the problem formulation. For the purpose of classifying scheduling problems, therefore, it is important to distinguish between two types of systems: tightly coupled and loosely coupled. Tightly coupled systems use shared memory to communicate between

processors, and communication times can be considered as negligible at all times. In loosely coupled systems, on the other hand, each processor has its own memory and communication must be done via some form of data bus or switching network. In the most loosely coupled system of all—the computer network—each processor "node" is a completely independent unit and communication is via external cables or telephone hookups. In the following discussion, loosely-coupled systems are assumed.

Important for the determination of optimal schedules are whether direct communication is possible between any two processors and whether there is the possibility of contention among messages for the use of the communication channels. In the ideal case—complete contention-free connection between all processors—the calculation of communication delays is relatively easy, while in a partially connected system with shared busses, prediction of exact communication delays may be impossible. Similarly, if the average communication delays are extremely small in comparison to the average computation time of the tasks, the effect of these delays will be minor in terms of choosing a good schedule, whereas if the communication delays are much greater than the average computation times, then planning to minimize these delays may be more significant than worrying about intelligent distribution of the task workload. In the two extreme cases—zero communication delays and infinite communication delays—the scheduling problem reduces to classical multiprocessor scheduling: in the latter case, to the scheduling of independent tasks. The specific assumptions needed on the communication between tasks in such systems are presented in Chapter III.

Other characteristics of the architecture also play a role in the determination of optimal schedules: for example, the relative speeds of the different processors, whether or not the processors are equivalent in terms of the jobs they can perform, and what kind of control of the processors is possible. This last characteristic leads to a gross classification of multiple processor computing systems according to the

amount of independence of control. Flynn [FLYN66] proposed the widely accepted acronyms SISD (Single Instruction Single Data), SIMD (Single Instruction Multiple Data), and MIMD (Multiple Instruction Multiple Data) for increasingly generalized systems. SISD systems are the equivalent of single processor systems. SIMD systems, such as vector processors, apply the same operations to different data streams, allowing efficient parallel processing of large numbers of similar calculations. Finally, in the MIMD systems, control of the processors is independent, allowing each processor to apply its own set of instructions to its own data stream. In this dissertation, the following assumptions are universally observed:

- (1) The computer system is an MIMD system, tightly or loosely coupled according to the problems being discussed, and
- (2) All processors are assumed identical: they operate at the same speed and any task can be performed equally well on any of the processors.

1.3. Concurrent Programming

In order to understand the significance of the problem of improving the average turnaround time as presented in Chapter II, it is necessary to understand the idea of concurrent programming. The normal high-level programming languages allow the user to write very sophisticated programs, but all such programs share the property of being sequential: they are to be executed in a predetermined order, one instruction at a time. In a single-processor environment, this is perfectly natural, but in a multi-processor environment it is too restrictive. Concurrent programming makes use of hardware and system software in such a way as to allow the simultaneous execution of segments of a program which are independent of one another logically. High-level language constructs such as FORK-JOIN and COBEGIN-COEND support

user-specified concurrency, while optimizing compilers written especially for particular systems may locate and exploit implicit concurrencies in a program written sequentially.

When a multiple processor system (MPS) is used to support a multi-programming environment, and if the individual programs are written as concurrent programs, then the collection of tasks available for scheduling at any moment breaks into a number of subsets, each subset belonging to a specific program or "job." If we are interested in average turnaround time as a performance measure, from the user's point of view it is not the turnaround time of the individual tasks but that of the complete jobs which is of interest. When a program is run, there is no particular interest in how soon a given subroutine finishes, but rather in how soon the whole program finishes. This theme is developed further in Chapter II.

1.4. Performance Analysis

In designing computers, operating systems, compilers, and other system tools, it is often necessary to evaluate the relative performance of one system versus another or versus some standard. Such evaluation falls into the general area of performance analysis. A wide range of techniques, such as benchmarking, simulation, figures of merit, and others, is used depending on the particular situation. An important first step, however, is deciding exactly what aspects of the system's performance to measure and under what criteria. As observed in the discussions of scheduling above, there are many—often conflicting—goals a scheduler may have: the same is true of other aspects of the system. While high counts of instructions per second may be a respectable goal, achieving this goal through inefficient code or only for computation-intensive programs may not really indicate high performance. With the understanding, then, that there are many kinds of performance and many ways to

evaluate each kind, the succeeding paragraphs discuss some of the performance measures relevant to scheduling.

Perhaps the most basic measure is that of system throughput, which is usually measured in jobs completed per unit time. Throughput is therefore a measure of how much useful work a computing system is performing in a given time interval. As a basis for comparison of performance, however, throughput can be misleading unless like-sized jobs are used in making the comparison. Ideally, in order to compare the throughput of two scheduling policies, they would be tested on the same set of jobs or on jobs with very similar characteristics.

The throughput is easily calculated for a dynamic scheduling situation: a related measure for static scheduling is the makespan, or total time required to complete a given set of jobs. In the static case, in fact, the throughput is essentially the reciprocal of the makespan. Both of these criteria measure the same "quality" of system performance; neither, on the other hand, relates to the satisfaction of an individual user in terms of the time required for his job to be completed. Scheduling policies favoring high throughput (low makespan) tend to place long jobs first or run jobs in first-come-first-served (FCFS) order, unduly lengthening the time required to complete many shorter jobs.

Minimizing the average turnaround time of jobs in the system is a quite different kind of performance goal, closely allied to the goal of user satisfaction. Improving the turnaround time of the jobs in the system, unfortunately, may not improve throughput at the same time. The turnaround time of a job is defined as the time from submission of the job to the time of completion. This is also frequently referred to as the flow time of the job. For static scheduling problems, the submission time of all jobs is taken as $t = 0$, so the average turnaround time is just the average completion time. Note also that minimizing the average turnaround time is the same as minimizing the total flow time--the sum of the flow times of all

the jobs. Surprisingly, running the shortest job first optimizes turnaround time while, at least on two processors, running the longest job first reduces the makespan and hence improves the throughput. In general, in an unsaturated (or under-utilized) dynamic scheduling situation, scheduling has little effect on throughput but can improve turnaround time, while in a saturated (or over-utilized) system, these two criteria tend to be opposed to one another [KRUC78, p. 533].

The response time of a system is often used as a performance measure, particularly in real-time installations such as interactive systems and control systems. This criterion is usually defined as the time from job submission to the beginning of the first output produced by the job, but variations on this definition also appear. The response time is meant to measure how long a user or external input source must wait from the time of input to the time it receives some response to its input. It is therefore, like turnaround time, related to user satisfaction, or, in the case of time-critical control, to the usefulness of the system. Response time removes some of the dependency on computational speed that the turnaround measure has, and is related more directly to another measure: waiting time. The waiting time of a job is the amount of time it spends in the "wait queue," or, more precisely, the amount of time from arrival to completion that the job is ready for processing but not being processed. In most classical scheduling problems, the relationship

$$\text{turnaround time} = \text{waiting time} + \text{processing time}$$

is assumed to hold, but if I/O time is considered as a third status, then the "=" becomes "≥." Moreover, if a job can be concurrently scheduled on more than one processor, the turnaround time can be less than the processing time!

Rather than looking at the jobs themselves, performance can also be measured by CPU utilization. This is normally expressed as the percent or fraction of the total time that the CPU is kept busy. For multiprocessor systems, this is measured individually for each processor or as the average over the processors. For single-processor

systems, this is not a useful criterion for the evaluation of scheduling policies since it is more related to the demand placed on the system and the degree of multiprogramming maintained than on the method of ordering the job executions. On the other hand, "load balancing" techniques for multiple-processor systems work to equalize the utilization of the various processors and are frequently treated as scheduling techniques. In this dissertation load balancing and CPU utilization are not considered as scheduling objectives.

A final performance criterion, speed-up, is not an absolute measure of performance but rather a term applied to the improvement achieved by a MPS over a single processor system. Speedup is the result of a number of parameters, the most important of which are the number of processors, the type of jobs being run, the characteristics of the interconnections between processors, and the scheduling policy used. Usually speedup is applied to one of these parameters at a time, holding the others constant, so as to compare the speedup of two systems, the speedup of two competing algorithms, or the speedup attained by two different scheduling policies. An appropriate definition of speedup is

$$S = \frac{\text{sequential processing time}}{\text{concurrent processing time}}$$

where, if it is the scheduling policy that is being investigated, it is assumed that the same jobs are run on the single- and multiple- processor systems.

This chapter has given some general ideas about scheduling and the kinds of systems the scheduling methods apply to. The next two chapters take up the specific scheduling problems which are the main focus and source of results of this dissertation. The final chapter broadens the view again, considering a wide range of possible extensions.

CHAPTER II

TURNAROUND TIME IN A GENERAL PURPOSE MPS

One measure of effective use of a multiuser system is the average turnaround time of the jobs in the system. If jobs are indivisible units, the venerable "Shortest Job First" (SJF) strategy is the best that can be done [CONW67]; however, for concurrent programs which have parts which can be run simultaneously on different processors, this strategy is no longer optimal. The significant point is that the objective of improving the average turnaround time of the jobs in the system is not achieved by improving the average turnaround time of the tasks which form the (possibly) concurrent pieces of the job. This chapter is devoted to the study of the problem of minimizing the average turnaround time for collections of concurrent programs running on a multiprocessor system.

As discussed at the end of Chapter I, lowering the average turnaround time is an objective which favors user satisfaction, since an individual user of a multiuser system is interested in the time from submission to completion of her job, not in how many jobs the computer can complete in that time or even whether it completed any other jobs. It is also closely related to the effective use of computing resources, since each job may tie up a number of peripheral devices or files while it is running, making other jobs wait for their release.

This chapter presents research on the problem of minimizing the average turnaround time of jobs consisting of precedence-related tasks as described in the first paragraph. The problem is attacked in two ways. First, best-case and worst-case bounds are provided for the most obvious extension of the usual SJF algorithm for static scheduling. Second, a scheduling simulator and its results are discussed in

order to compare a number of heuristic dynamic scheduling policies. Section 2.1 compares the dynamic and static scheduling problems; Section 2.2 presents the best- and worst-case bounds analysis, while the following two sections present the simulation method and results. Section 2.5 takes a deeper look into the way in which the size of a job can be measured. It is shown that a new definition of job size may provide a good basis for an improved scheduling heuristic. A summary is provided in Section 2.6.

2.1. Dynamic and Static Scheduling Problems

The static version of a scheduling problem assumes that a collection of jobs is given and available at the outset and that a complete schedule could be created at that time. The dynamic version assumes that the jobs arrive at different times and that scheduling must be done in real time along with the running of the jobs. In the case of the traditional problem of minimizing the average turnaround time of a number of indivisible jobs (the non-preemptive case), if the jobs are all independent, then the static problem is solved by the multi-processor version of Shortest Job First (SJF, also known as SPT for Shortest Processing Time) [BAKE74a]. This orders the jobs from shortest to longest and schedules the next available job in that order whenever a processor becomes available. If there is any type of precedence relation among the jobs, the problem is NP-hard unless there are no more than two processors and all jobs are unit time [LAGE81b].

The dynamic version of the problem is solved for independent jobs in the preemptive case on a single processor by the related Shortest Remaining Time First policy (SRTF), which is the preemptive version of SJF [CONW67]. For m processors, even this method can fail unless all jobs are available at the same time [MART77]. SRTF guarantees that all processors are busy when possible and the

remaining processing time of any waiting job is longer than the remaining processing time of any job being run. The non-preemptive dynamic case has no "solution" even on a single processor in the sense that no matter what policy is adopted, sometimes it would have been better to wait, leaving a processor idle for a small interval of time, for the arrival of another job and scheduling it next rather than any of the jobs already available. It is obvious that no scheduling policy can be smart enough to know when to wait for a future event. Nonetheless, SJF is asymptotically optimal for many possible distributions of arrivals and processing times [AGRA84]. •

In a static problem closely related to the dynamic one, all information is available at the outset, but the jobs have "release" times indicating the earliest allowable start time for each [DEOG83]. If all jobs have length one, there is a polynomial-time solution for any number of processors, as discovered by Lawler [LAWL64]. Interestingly, the same problem on processors with differing speeds was still open according to the 1981 classification of [LAGE81b]. As mentioned above, in the preemptive case, SRTF is useful but not always optimal. Martin-Vega and Ratliff [MART77] point out that SRTF does, in fact, maximize the makespan!

Turning to the central problem of this chapter, the exact scheduling problem to be studied must be made precise. The general assumptions presented in Section 1.1 hold here as throughout the remainder of the dissertation. The objective of the scheduling is to minimize the average turnaround time of a set of independent jobs, or equivalently, to minimize the total flow time of the set of jobs. Therefore, the problem appears to fall into the class of problems given in Section 1.1 as $P/o/\sum C_j$, where the "o" standing for the empty precedence relation.

The novelty here, however, is that each job is considered to consist of a number of non-preemptable "tasks," which are related to one another by a precedence relation, called " $\rightarrow$." This task structure not only allows the job to be preempted between tasks but also to have two or more of its tasks run—perhaps concurrently—on

different processors. In other words, the schedulable units are the tasks rather than the jobs. The effect of the precedence relation is to restrict the order in which the tasks may be executed: $T \rightarrow T'$, or T precedes T' implies that T must be completely executed before T' can start. T is called an "immediate predecessor" of T' and T' an "immediate successor" of T , written $T \rightarrow_I T'$, if $T \rightarrow T'$ and there is no further task T^* such that $T \rightarrow T^* \rightarrow T$. A precedence relation is often given by the directed acyclic graph (DAG) of the immediate successor relation, in which the nodes are the tasks and an arrow is drawn from T to T' if and only if $T \rightarrow_I T'$.

As an artificial illustration, suppose that a job consists of tasks labeled $T_2, T_3, \dots, T_8$ and a precedence relation which satisfies the condition that $T_i \rightarrow_I T_j$ if and only if i divides j or $j+1$. The corresponding DAG is then the one shown in Figure 2.1.

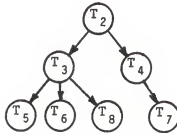


Figure 2.1. A Sample Precedence DAG

Notice that $T_2 \rightarrow T_8$ but it is not true that $T_2 \rightarrow_I T_8$: hence no arrow is drawn from T_2 to T_8 .

With the introduction of the task structure within each job, the resulting scheduling problem for minimizing the average turnaround time of the jobs no longer falls into the classification scheme; in fact, the problem does not appear to have been dealt with in the literature before. We introduce here the notation

$$P / \text{internal-prec} / \sum C_j \quad (1)$$

to represent this problem as an extension to the notation of [LAGE81b] discussed above. In order to make it clear that minimizing the average turnaround time of a collection of such jobs is not the same as minimizing the average turnaround time of all of the tasks that make up the jobs, consider the following simple example to be scheduled on two processors.

$$J_1 = T_1: 1 \quad T_2: 4 \quad J_2 = T_3: 2 \quad T_4: 2 \quad (2)$$

Here, the number following each task is the required processing time, and the absence of arrows indicates that the precedence relations are empty. Even in this very simple case it can be seen that the Gantt chart of Figure 2.2(a) gives an optimal schedule for minimizing the average turnaround time (ATT) of the tasks (giving a value of three), whereas Figure 2.2(b) obtains a better average turnaround time for the jobs (a value of four as opposed to that of 4.5 for Figure 2.2(a)).

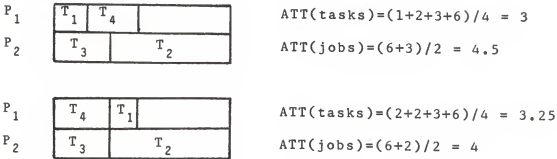


Figure 2.2. Job Versus Task Turnaround Time

Due to the known results about traditional scheduling problems, it is easily seen that the static problem presented here is NP-hard, and this is stated in the first theorem.

THEOREM 2.1: The problem $P/\text{internal-prec}/\sum C_j$ is NP-hard.

PROOF: In the simple case of a single job, minimizing the turnaround time is the same as minimizing the makespan. A single job in the given problem, however, consists of tasks in an arbitrary precedence relation, and this problem—even with all unit-time tasks—was shown by Ullman in 1975 [ULLM75] to be NP-hard. Therefore the given problem, being clearly more difficult than a particular case, is NP-hard. $\square$

For the static problem, the following section obtains a best-case lower bound on the time required by any schedule and obtains a possible worst-case upper bound on the time required by an extended version of SJF. For the dynamic (non-preemptive) problem, as in the traditional case, there can be no truly optimal algorithm due to the random arrivals, but succeeding sections present the results of simulation work comparing a variety of heuristic algorithms intended to lower average turnaround time.

2.2. Analytic Approach to Heuristic Algorithms

In the case of problems which are known to be NP-hard, the only practical recourse is to find suitable heuristic scheduling methods which produce suboptimal but reasonable schedules. Many such heuristic methods have been suggested and implemented, and many comparative studies of such methods have been done, particularly by industrial engineers concerned with factory machine scheduling problems [RUSS84]. It has been shown in simulations, as well as in actual applications, that the SJF discipline outperforms other reasonable heuristics consistently in a wide variety of situations where it is no longer optimal (always with the objective of minimizing average turnaround time) [BRUN81, BAKE74a, CONW67, DEOG83]. Analytic methods have also been applied to show, for example, that SJF has the optimal expected turnaround time in a nondeterministic system with exponentially

distributed arrival times and service times [BRUN81]. This is an example of "average case analysis."

With some other scheduling problems, such as minimizing the makespan of a number of independent tasks, "worst case analysis" has been applied to establish upper bounds on performance. The best known such result is that of R. L. Graham, who showed that the Longest Job First (LJF) strategy is never worse than 1.333 times that of an optimal strategy [GRAH69]. Later work has produced more sophisticated algorithms with better worst-case bounds, such as the Multifit algorithm of Coffman, Garey, and Johnson [COFF78].

In this chapter certain heuristic strategies based on the SJF and SRTF algorithms are discussed and simulation is used to compare them against each other and against a "random" scheduling strategy. The simulator assumptions and program are discussed and the results analyzed here. The heuristic strategies tested are all "two-level" strategies which use one method to select the job to be run next and another to select the task within that job.

To begin with, an analysis is made of an extended version of the SJF algorithm as a way of motivating its use as a heuristic strategy for the problem in hand and of gaining some perspective on its efficacy.

Start with a collection of jobs $J_1, J_2, \dots, J_n$, and suppose that each job, J_k , consists of a number of tasks,

$$T_{k,1}, T_{k,2}, \dots, T_{k,r_k}$$

related by a precedence relationship in the form of a DAG. Let u_k be the total processing time of J_k ; in other words, u_k is the sum of the processing times of the tasks $T_{k,i}$. Finally, assume that the processing times are in increasing order:

$$u_1 < u_2 < \dots < u_n.$$

The scheduler may only schedule a task to be performed if all its predecessors have

finished. The objective is to minimize the total flow time, or the sum of the turnaround times of the jobs (not tasks). We further assume m identical processors $P_1, P_2, \dots, P_m$. As always, any task may be scheduled on any processor. The first algorithm is the classical SJF, called here Sequential SJF for emphasis.

Algorithm 2.1. SSJF (sequential shortest job first)

Whenever a processor is free, assign the next job in numeric order (hence in size order) to that processor and run it to completion without interruption. If more than one processor is available at the same time, assign to the lowest numbered processor first.

Algorithm 2.2. CSJF (concurrent shortest job first)

Whenever a processor is free, assign a ready task from the next job in numeric order to that processor and run the task to completion. If more than one task from the lowest numbered job is ready at the same time, choose the lowest numbered task. (This is referred to as First Available Task (FAT) in Section 2.4.) If more than one processor is available at the same time, assign to the lowest numbered processor first.

LEMMA 2.1: With SSJF, job J_k with $k = qm + r$ is assigned to processor r and scheduled to start at time

$$s_k = \begin{cases} 0 & \text{for } k \leq m \\ u_r + u_{r+m} + \dots + u_{r+(q-1)m} & \text{for } k > m. \end{cases}$$

PROOF: For the first m jobs, this is obvious. Since the jobs are listed in non-decreasing order, the times that the processors next become available also form a non-decreasing sequence as the processor number varies from 1 to m . Therefore, SSJF assigns the $(m+1)$ -st job to P_1 to start when J_1 finishes: time u_1 . Similarly,

the next job is assigned to P_2 and so on, cycling through the processors in numeric order. The start times are calculated easily as the sum of the processing times of the jobs previously assigned to the corresponding processors. $\square$

LEMMA 2.2: (1) SSJF and CSJF never leave a processor idle until all jobs have been scheduled. More precisely, if a processor is idle on some interval of time $[t, t']$, then all jobs were scheduled to start no later than time t .

(2) CSJF never leaves a processor idle unless all unfinished jobs are running. More precisely, if a processor is idle on some interval of time $[t, t']$, then every unfinished job has at least one task running on that interval (or finishes during the interval).

PROOF: (1) This is an easy consequence of the way that the algorithms operate: whenever a processor becomes free, something is immediately scheduled on it unless there is nothing left to schedule.

(2) If some unfinished job is not running, then it must have at least one ready task. Therefore as soon as a processor becomes free, that task (or some other) will be scheduled on it; in other words, any time there is an idle processor there can be no unfinished jobs with no tasks running. $\square$

LEMMA 2.3: Call a job active at time t if it was started before time t but has not finished by time t . Under the CSJF scheduling policy, at any time t there can be no more than m active jobs.

PROOF: Suppose there are $m+1$ active jobs at time t . Let J be the highest numbered (longest) of these jobs, and hence the latest one to start. Let $s(J) < t$ be the time it was scheduled to start. At that time the other m lower-numbered (shorter) jobs were already active, so none of them could have had a task ready at time $s(J)$. But the only reasons that an unfinished job has no task ready are that all tasks are already scheduled or that each unscheduled task has some predecessor

currently running. In either case, some task of each of the unfinished jobs must have been running at time $s(J)$. This is a contradiction to the fact that there are only m processors and there were at least $m+1$ jobs (counting job J) running at time $s(J)$. Therefore no such set of more than m jobs can exist. $\square$

LEMMA 2.4: At any time t before all jobs are completed and for any k between one and the minimum of m and the number of unfinished jobs, under the CSJF scheduling policy there must be k processors busy running only the k lowest numbered (shortest) unfinished jobs.

PROOF: The lowest numbered unfinished job has the highest priority. Therefore once it gains this status, no other job can preempt it and it will always be running on some processor. (Whenever one of its tasks finishes either another will be ready or another will still be running on another processor.) Therefore the lemma is true for $k=1$. Assume that it is true for $k-1$, for some $1 < k \leq m$, and that there are at least k unfinished jobs at time t .

By the induction hypothesis there are $k-1$ processors which are running only the $k-1$ lowest numbered unfinished jobs at time t . Let J be the k -th lowest numbered unfinished job at that time. If J is also running on some processor at time t , then the lemma is true for case k . If, on the other hand, J has not yet been started, then the only jobs running on all m processors must be lower numbered than J . Since CSJF does not leave processors idle as long as there are jobs that have not yet started (Lemma 2.2), there are $m \geq k$ processors running the first $k-1$ unfinished jobs. Finally, if J has already been started, but J is not running at time t , let t_1 be the last time before t that a task of J was completed. By the induction hypothesis, at any time before t_1 there must be $k-1$ processors running the $k-1$ lowest numbered unfinished jobs. Since from t_1 to t no task of J is running even though J is unfinished at this time, any processor that became free must have been occupied by a

task from a job of lower number. Moreover, no processor can be left idle during this interval since at least job J has a ready task at this time. In particular, the processor running J and the other $k-1$ processors running lower numbered jobs must all continue to run jobs numbered lower than J from t_1 to t . At time t , the same argument prevails since there are still at least k unfinished jobs and J is not running. Therefore the lemma is true for k , and by induction it is true for all specified cases.

□

LEMMA 2.5: The total flow time of the first k jobs under CSJF is no more than the total flow time of the first k jobs under SSJF for all $k = 1, 2, \dots, n$.

PROOF: The flow time (or turnaround time), $ft(J)$ of a particular job J is given by $ft(J) = s(J) + (\text{time that } J \text{ is running}) + (\text{time that } J \text{ is active but not running})$. We argue that for each of these three terms, if J is doing worse than it would under SSJF, then some other lower numbered jobs are doing better.

Under SSJF, the time that J is running must be just $u(J)$, the total processing time, while under any policy it cannot be more. Therefore, CSJF does as well or better on this term.

Under SSJF, the job J is never active and not running since each job is run without interruption once started. If, under CSJF, an active job J is not running on some interval $[t, t']$, then by Lemma 2.4, there must be k processors busy running the k lowest numbered jobs, where J is the k -th lowest. But since J itself is not running, the k processors are running only $k-1$ jobs. Consequently, at any given time in the interval, one of those lower-numbered jobs must be running on two processors. This means that while $ft(J)$ is lengthened by $t'-t$, the turnaround times of some other lower-numbered jobs must be shortened by a total amount of at least $t'-t$. Therefore, CSJF is doing at least as well as SSJF.

Now let $S(J)$ be the start time of J under SSJF and suppose that J is the lowest numbered job such that $s(J) > S(J)$. Then up to time $s(J)$ at least job J had not started, so by Lemma 2.2 no processors were idle before this time; hence, the system under CSJF has dedicated

$$m \times s(J) \quad (3)$$

units of processing time to running the jobs numbered lower than job J by time $s(J)$. On the other hand, after time $S(J)$, the policy SSJF is using one processor to run job J and hence is dedicating at most

$$m \times S(J) + (m-1) \times (s(J) - S(J)) = S(J) + (m-1) \times s(J) \quad (4)$$

units of processing time to these jobs by time $s(J)$. In other words, CSJF put in $s(J) - S(J)$ extra units of processing time running the lower-numbered jobs before starting J . Let J' be a lower numbered job which received, say, dt more units of processing under CSJF than under SSJF by time $s(J)$. Then at time $s(J)$, J' has dt less time to go under CSJF unless it is delayed after time $s(J)$. As observed in the foregoing paragraph of this proof, however, such a delay cannot cause any increase in the total active time of the jobs up to and including job J' . Moreover, by the minimality of J , all of the jobs up through J' start at least as early under CSJF as under SSJF. Therefore, the total flow time of the jobs up through J' is at least dt less under CSJF. Since a total of $s(J) - S(J)$ extra time units were received by the jobs before J , the total flow time under CSJF of the jobs before J is at least $s(J) - S(J)$ less than under SSJF, and therefore the total flow time including job J is no more than that under SSJF. $\square$

THEOREM 2.2: The total flow time achieved by CSJF is at least as small as that achieved by SSJF.

PROOF: This is an immediate consequence of Lemma 2.5 taking $k = n$. $\square$

Turning now to obtaining a worst case bound for CSJF, Theorem 2.2 assures that the flow time of SSJF can be used. The following well-known result [BAKE74a] indicates that this flow time is easily calculated.

LEMMA 2.6: The total flow time of n jobs run on m processors under SSJF is given by

$$FT = \sum_{i=1}^n \left\lceil \frac{n-i+1}{m} \right\rceil u_i, \quad (5)$$

where the symbol $\lceil a \rceil$ means the least integer greater or equal to a .

PROOF:

$$\begin{aligned} FT &= \sum_{i=1}^n (s_i + u_i) \\ &= \sum_{i=1}^n \sum_{k=i \pmod m, 0 < k \leq i} u_k \quad \{\text{by Lemma 2.1}\}. \end{aligned}$$

Each u_k appears in this expression as many times as there are integers between k and n which are congruent to k modulo m . These are just the numbers $k, k+m, k+2m, \dots$, and there are $\lceil (n-k+1)/m \rceil$ of them, proving the lemma. $\square$

COROLLARY TO THEOREM 2.2: The total flow time of n jobs run on m processors under CSJF is never more than

$$\sum_{i=1}^n \left\lceil \frac{n-i+1}{m} \right\rceil u_i.$$

THEOREM 2.3: Let ft be the flow time for n jobs on m processors under CSJF and let ft_{opt} be the optimal flow time for the same set of jobs under the optimal schedule. Then

$$\frac{ft}{ft_{opt}} \leq 1 + \frac{\sum_{i=1}^n p_i u_i}{\sum_{i=1}^n (n-i+1) u_i}, \quad (6)$$

where $p_i = m -$ (the remainder of dividing $n - i + 1$ by m) if this remainder is non-zero, and $p_i = 0$ if the remainder is 0.

PROOF: The best possible case for scheduling n jobs on m processors occurs when each job J_i consists of m equal-sized tasks of length u_i/m which can be scheduled to run concurrently. Then the optimal flow time would be

$$ft_{uni\ opt} = \sum_{i=1}^n (n-i+1) u_i / m. \quad (7)$$

In general, then, $ft_{uni\ opt} \leq ft_{opt} \leq ft \leq FT$, the flow time under SSJF, and

$$\frac{ft}{ft_{opt}} \leq \frac{ft}{ft_{uni\ opt}} \leq \frac{\sum_{i=1}^n \lceil n-i+1 \rceil u_i}{\sum_{i=1}^n (n-i+1) u_i / m}. \quad (8)$$

Observe that if

$$a = mq + r \text{ with } r < a,$$

then

$$\left\lceil \frac{a}{m} \right\rceil = \begin{cases} \left\lceil q + r/m \right\rceil = q & \text{if } r = 0 \\ q + 1 & \text{otherwise.} \end{cases}$$

Therefore, $m \lceil a/m \rceil = mq$ or $mq + m$, which is the same as a (if $r = 0$) or $a + m - r$ (if $r > 0$). By multiplying numerator and denominator of (8) by m and applying the preceding observation with $a = n - i + 1$,

$$\frac{ft}{ft_{opt}} \leq \frac{\sum_{i=1}^n ((n-i+1)+p_i)u_i}{\sum_{i=1}^n (n-i+1)u_i} = 1 + \frac{\sum_{i=1}^n p_i u_i}{\sum_{i=1}^n (n-i+1)u_i}. \quad \square$$

A few comments are in order to interpret this result. If all the u_i are equal, then the CSJF picks the next job more or less arbitrarily; therefore, how well it performs depends on how it schedules the tasks of each job. In this case the inequality of Theorem 2.3 becomes

$$\frac{ft}{ft_{opt}} \leq 1 + \frac{\sum p_i}{\sum (n-i+1)} = \frac{\sum p_i}{\sum i}.$$

If n is an exact multiple of m , say $n=rm$, then this becomes

$$\begin{aligned} ft/ft_{opt} &\leq 1 + r \times \frac{(0+1+\dots+(m-1))}{n(n+1)/2} \\ &= \frac{rm(m-1)}{n(n+1)} \\ &= 1 + \frac{(m-1)}{(n+1)}. \end{aligned} \quad (9)$$

If no assumption is made on the sizes of the jobs, then it is still possible to write

$$\begin{aligned} \frac{ft}{ft_{opt}} &\leq 1 + \frac{\sum p_i u_n}{\sum (n-i+1)u_1} \\ &= 1 + (m-1)u_n/(n+1)u_1. \end{aligned} \quad (10)$$

If at least

$$\frac{u_n}{u_1} < \frac{(n+1)}{(m-1)},$$

then $ft/ft_{opt} \leq 2$, and the ratio tends to 1 if n gets much larger than m while u_n/u_1 remains constant. Looking at it another way:

$$\begin{aligned}
\frac{ft}{ft_{opt}} &\leq 1 + \frac{\max\{p_i\} \times \sum u_i}{\min\{i\} \times \sum u_i} \\
&= 1 + \frac{m-1}{1} \\
&= m.
\end{aligned} \tag{11}$$

Therefore, combining (10) and (11) gives

COROLLARY TO THEOREM 2.3:

$$\frac{ft}{ft_{opt}} \leq 1 + (m-1) \times \min\left\{1, \frac{u_n}{(n+1)u_1}\right\}.$$

2.3. Multiprocessor Simulator

This section describes the simulator developed in order to compare a number of different heuristic scheduling strategies in an environment such as that described in Section 2.1. The simulator consists mainly of a driver program which acts like a multiprocessor system of hardware and appropriate interrupts, a high-level scheduler which determines admission into the system of new jobs, and a number of interchangeable dispatchers which embody the different scheduling strategies. Whereas the high-level scheduler reads job information from a job file and initializes the appropriate data structures containing the necessary job information, the dispatcher is capable of scanning the job "queue" and selecting the appropriate task according to the given discipline. The dispatcher also updates job and task information and informs the high-level scheduler when a job is completed. Two undergraduate students at the University of Florida, Dennis Suppe and Borden Wilson, assisted in programming the simulator [SUPP86, WILS86]. The actual Pascal code appears in [WILS86].

There are a number of design decisions which critically affect the results of the simulations. First, it is assumed that the scheduling itself contributes no overhead.

Thus the system maintains a "clock" for each processor which is simply updated by the length of each task scheduled on that processor. The turnaround time, or flow time, of each job is then calculated as "time completed - time entered system." Second, the high-level scheduler maintains a constant degree of multiprogramming (DM) as long as there are more jobs in the input file. This means that at the start of the simulation a value for the DM is chosen and the simulator enters DM jobs into the system at time zero. Henceforth, whenever a job is finished, the scheduler reads a new job with starting time equal to the finish time of the job just terminated. Once the file is emptied of jobs, the simulation continues until all the jobs remaining in the system are completed.

A third element of the design of the simulator is that at all times the scheduler has at its disposal all the unscheduled tasks of all the jobs in the system, together with the necessary information to implement the algorithms described below. In particular, the scheduler must know the total processing time ($TPT = u(J)$) of each job, the processing time of each remaining task, the precedence relations among the tasks, which tasks belong to which jobs, and, for some of the algorithms, additional information. All of this information is stored within a number of two-dimensional matrices—one for each job in the system. The matrix is essentially an adjacency matrix for the DAG representing the precedence relation of the job, where the (i, j) entry is one if task T_i is an immediate predecessor of task T_j and is zero otherwise. This matrix is modified, however, in a number of ways. The diagonal—or (i, i) —entries are set equal to the processing times of the corresponding tasks. The tasks are always numbered such that if T_i precedes T_j , then $i < j$. This guarantees that no entries of 1 will appear below the diagonal, and therefore these entries can be used to store other information about the job. Finally, the matrix is actually augmented by an extra row and column, so that if there are (at most) ten tasks, then the matrix has subscripts running from zero to eleven. A one in the $(0, j)$ entry, for

example, indicates that task j is available for scheduling (not yet scheduled but no unscheduled predecessors).

Besides the simulator program itself, two other important auxiliary programs have been developed: a job pool generator which constructs files of jobs with various characteristics and a statistical analysis package which analyzes the output of the simulator to give information on the relative performance of the various dispatchers.

2.4. Simulation Results

The class of algorithms being simulated can be described as "bilevel." These algorithms use one criterion for the selection of the next job to be scheduled and a different criterion for the selection of the specific task within the chosen job. This general strategy results from the necessity to order the execution of whole jobs while at the same time selecting a distribution of the tasks within each job to complete the chosen jobs as quickly as possible. Specific algorithms are created by combining one of the "job-level" strategies with one of the "task-level" strategies.

All of the "intelligent" strategies selected for testing are based on the intuitive idea of running the smallest jobs first and doing so as quickly as possible. Two related but different measures of the "smallness" of the job are used: the total processing time (TPT) of the job and the critical path time (CPT) of the job. The TPT of a job J_i has been represented by u_i in this chapter, whereas CPT is the time required to execute the longest chain of tasks in the job. Effectively, CPT gives a minimum time required to run the job on any number of processors, while TPT is the time the job would take to run on one processor without interruptions. Once a job is chosen according to one of these criteria, a simple method of choosing an appropriate task is to select that task with the most immediate successors. This tends to provide as many tasks as possible available at any given moment and

therefore allow as many processors as possible to cooperate in finishing the job quickly. Another approach is to run the task heading the longest remaining chain of unscheduled tasks. Combining these methods with other related ones and some "unintelligent" ones produces the following list of possibilities.

1. Job Level

- a. SJF Select first the job with the shortest total processing time (the sum of all the task processing times).
- b. SCPP Select first the job with the shortest critical path (length of the longest chain of tasks).
- c. SRTF Select first the job with the shortest remaining processing time.
- d. SRCPP Select first the job with the shortest remaining critical path.
- e. Random Select a random job.

2. Task Level

- a. MISF Select first the task with the most immediate successors.
- b. LRTF Select the task heading the longest sequential chain of remaining tasks.
- c. FAT Select the first (lowest numbered) available task.

Just how "good" each of these methods might prove to be appears to be dependent on some of the characteristics of the DAGs of the typical jobs being scheduled. For example, jobs which consist of a large number of small, independent tasks may take a long time to complete even though they have very short CPTs. Conversely, jobs which are almost entirely sequential may take longer to finish than others with larger TPTs but exhibiting more concurrency. It was therefore decided to run the simulations on three different type of job pools, all with approximately the same average number of tasks: "Wide Jobs," with small CPT, "Long Jobs," with most

tasks lying along the critical path, and "Random Jobs," with DAGs generated randomly.

Before running the simulations, a number of hypotheses were made about the expected effects of the different parameters on the turnaround times expected. These were as follows:

- H1. As the number of processors increases, the differences between the scheduling policies will decrease, since if there are enough processors, no ready task has to wait and all reasonable scheduling policies produce the same results.
- H2. With higher degrees of multiprogramming, the differences between the policies would be more apparent since at each moment the dispatcher has more jobs to choose from and hence the choice is more critical.
- H3. The SJF and SRTF strategies should, in general, outperform the SCPF and SRCPF with the "wide" jobs, because when there are many tasks but short critical paths, the length of the critical path is a poor estimate of the time required to run the job. This should be all the more true when there are few processors.
- H4. The SCPF and SRCPF strategies should, in general, outperform the SJF and SRTF methods with the "long" jobs, because when a large number of the tasks lie along the critical path, the length of this path becomes the determining factor in how long it will take to finish the job. This should be even more pronounced when there are many processors.
- H5. The Random job-level strategy should do markedly worse than any of the other methods, except when there are many processors and a low degree of multiprogramming.

The simulator was run on the VAX 11/780 system of the University of Florida CIRCA system with job files of approximately 500 jobs. Each run matched a job file of given characteristics against a dispatcher using a certain strategy and a high-level scheduler maintaining a particular level of jobs in the system. Moreover, this was

done for several different numbers, m , of processors. As can be seen in Table 2.1, for the values chosen there are 840 possible different simulation "settings." Data were actually collected on all but a few of these possibilities.

TABLE 2.1. Factors Considered in Simulation

1. Degree of Multiprog.:	5, 10, 20, 30
2. Job Types:	Wide, Long, Random
3. Number of Processors:	3, 5, 7, 9, 11, 13, 15
4. Dispatcher	
Job Level:	SJF, SRTF, SCPF, SRCPF, Random
Task Level:	MISF, FAT

The actual simulation results appear in Wilson's work [WILS86]. Two kinds of statistical tests were applied to the data in order to check the significance of the results. The first performed tests on the hypothesis of the form $H_0: \mu_1 = \mu_2$ against the alternative $H_a: \mu_1 > \mu_2$, where the μ_i represent the average turnaround times of matched runs (equal levels of multiprogramming, numbers of processors, and job characteristics). This is a standard test of hypotheses for the equality of the means of two populations and is based on the use of a table of probabilities for the standard normal distribution (z -values). The sample variances are used to estimate the population variance. The details are reported in Ellis [ELLI86]. Appendix B shows the cases in which the alternative hypothesis of the form "Strategy A is better than Strategy B" could be accepted at the 90% confidence level. These tests of hypothesis were performed on the raw data consisting of all the individual turnaround times, and then these data were discarded [ELLI86]. (In total it comprised over two megabytes of storage.)

The second analysis was a post-facto application of the Friedman Two-way Analysis of Variance by Ranks [SIEG56]. This is a non-parametric test which was carried out for each fixed degree of multiprogramming and fixed number of processors. It tests the null hypothesis that all the five job-level scheduling methods

produce the same average turnaround times against the alternative hypothesis that there is a significant difference among these times. There are several reasons for the application of this procedure:

1. The tests of hypothesis carried out comparing two average turnaround times were done using the standard techniques and the z values of the standard normal distribution. Since the sample sizes were large (500), such tests should be relatively reliable, but two requirements were not satisfied: the standard deviations of the populations being compared were not at all equal, particularly when comparing the Random scheduler with one of the more intelligent ones; and the samples were not independent, since the different schedulers were run against the same input data. Thus further testing was required.
2. Since the raw data were not available, the test had to be run using each average turnaround time result as a single data item. Although such sample averages drawn from a given population are guaranteed by the Central Limit Theorem to have a normal distribution, averages corresponding to different settings of the independent variables have very different standard deviations. Moreover, unless data resulting from many different settings are lumped together, the sample sizes for further testing are quite small.
3. The Friedman Analysis of Variance method applies to
 - (a) data classified by rank only,
 - (b) data of unknown distribution and standard deviation,
 - (c) dependent (matched) samples, and
 - (d) testing for the equivalence of a number of means at the same time.

The approach chosen was then the following:

- A fixed value of m (processors) and DM (degree of multiprogramming) are chosen.

- The five average turnaround times obtained for the five different job-level scheduling methods with a fixed job type are treated as a matched set of data. The three sets—one for each job type—give five dependent samples of three values each ($k = 5$, $N = 3$) to which to apply the test.
- The Friedman test is a rank test, so the five matched values are replaced by their ranks—first through fifth—and the χ_r statistic is computed, where

$$\chi_r = \frac{12}{Nk(k+1)} \sum R_i^2 - 3N(k+1),$$

with R_i = the sum of the ranks of the i -th matched set. The χ_r value is then compared with the χ^2 value for the .10 level of significance (for $k-1 = 4$ degrees of freedom): $\chi^2 = 7.78$.

- This test is repeated for each of the seven values of m and four values of DM. This is all repeated for the two different task-level scheduling methods—MISF and FAT. The results showing significance at the .10 level appear in Appendix A.

Unfortunately, one of the most remarkable results is the generally small and unpredictable differences among the various strategies. Since each simulation run involved a large number of (over 500) simulated jobs being run, it was expected that there would be relatively clear "visual" differences in the results with the different scheduling policies. These differences did not materialize. Combining the results of the two methods of analysis yields the following conclusions:

1. Under the chosen design criteria and with the relatively small jobs (no more than ten tasks per job) used as data, the turnaround times are only marginally dependent on the scheduling strategy used.
2. Significant differences are more apparent with a low (5) degree of multiprogramming and with a high (≥ 9) number of processors, contrary to the expectations H1 and H2 above.

3. The only reasonably consistent finding was that the critical path methods outperform the shortest job methods (SJF and SRTF), particularly at small DMs and using the First Available Task task scheduler.
4. Due to combining of the data from the different job types in applying the Friedman test, no evidence for or against the hypotheses H3 and H4 above can be derived from that test. Notwithstanding, relatively strong support was found for H4 from the tests of hypothesis on two means. With long jobs, the critical path methods outperformed the other methods tested at a low degree of multiprogramming and with a middle to high number of processors. These results were found using the unintelligent task scheduler, FAT, and are corroborated by the Friedman Test.

In order to understand the low power of discrimination of these results, it is necessary to investigate the effect of the experimental design on the results obtained. First consider the effect of maintaining a constant degree of multiprogramming on the flow time. The flow time, ft , can be calculated as the sum of the individual finish times, f_i , of the jobs, J_i ; but it can also be calculated as the sum of the degree of multiprogramming (DM) times the time interval on which that degree is valid:

$$ft = \sum DM_i \times (t_{i+1} - t_i).$$

If DM is, in fact, constant, this just becomes $ft = DM \times \text{total time}$, independent of scheduling policy! At the end of each of the simulation runs, the remaining jobs are actually finished, dropping DM to zero. Thus, for example, if 504 jobs are run on 3 processors with 5 jobs in the system, then $DM = 5$ until 500 jobs have been run and then it drops to 4, then 3 and so on. This means that the flow time in this example is

$$ft = 5 \times (\text{makespan of the first 500 jobs completed}) + \\ 4 \times (f_{501} - f_{500}) + \dots + 1 \times (f_{504} - f_{503}).$$

Now any scheduling method that does not leave a processor idle unnecessarily can achieve no shorter makespan than $(\text{sum of } 500 \text{ smallest } u_i)/3$ and no longer makespan than $(\text{sum of } 499 \text{ largest } u_i)/3 + \text{largest } u_j$. The difference between these two is just

$$\text{largest } u_j + \text{sum of } (3 \text{ next largest } u_i - 4 \text{ smallest } u_i)/3.$$

The rest of the terms in ft comes to, at most, $10/3$ times the longest job processing time, u_j . All together,

$$ft - ft_{opt} < 13/3 \times \text{longest } u_j \\ + \text{sum of } (3 \text{ next largest } u_i - 4 \text{ smallest } u_i)/3.$$

For 500 jobs, this would amount to something like a 2% difference between the observed flow time and the optimal value and hence even less between two observed values.

In general, from the foregoing discussion it can be seen that the only effect that running a large sample of jobs (such as 500 jobs) has on the given bound on $ft - ft_{opt}$ is that the longest jobs may be longer and the shortest jobs shorter than would be the case with a small sample. This would not be the case if random arrival times were used in the simulation, since a variable degree of multiprogramming would be produced and hence, presumably, greater variability in the total flow times would be achieved by the different scheduling algorithms.

Another factor contributing to the homogeneity of the results might have been the way in which the data files were produced. First, 12 DAGs were created with the desired characteristics (long, wide, or randomly generated). Then 514 jobs were created by assigning exponentially distributed random processing times to the tasks of the DAGs, using the same 12 graphs repeatedly, each time with different processing times. Whereas the task times are exponentially distributed with mean and standard deviation of one, the processing times, $u(J)$, jobs with, say, ten tasks will be

approximately normally distributed with a mean of ten and a standard deviation of $1/\sqrt{10}$, or approximately .32. In other words, there is relatively little variation in the job sizes and hence less chance for the different policies to exhibit their powers.

2.5. Towards a Theory of Program Size

It is evident from looking at the scheduling strategies given in the previous section that the central idea to all of them (except the random strategy) is to select first the "smallest" job, in some sense of the word, and then to run that job as quickly as possible. The idea of "run as many of the jobs as quickly as possible" has strong appeal and is known in its guise of SJF to be optimal for non-preemptive static scheduling of nondecomposable jobs (jobs consisting of a single task). Nonetheless, it is not easy to specify exactly what is a "small" job when extending this idea to jobs which consist of a collection of precedence-related tasks which are to be run on several processors. Although the critical path time (CPT) and the total processing time (TPT) are well-worn measures of job size, neither one always tells us which job can be finished more quickly; however, we may conclude that the job's run time will be greater or equal to $S = \max\{ \text{CPT}, \text{TPT}/m \}$ if there are m processors. While this suggests using S as the measure of size, examples can be found to show that smallest "S" first is not an optimal strategy either.

One method to improving the scheduling strategy may be to design an easily calculable measure of job "size" which more closely identifies how long it should take to run the job on m identical processors. Such a measure can be obtained by extending the results established by Hu in his 1961 paper [HU 61], where the attention was restricted to unit-time tasks in an in-tree precedence graph. Following his ideas, we make the following definitions:

- (1) Assign a height, $h(T)$, to each task T in the precedence graph of a job J by setting
 - a) $h(T) = u(T)$ (the processing time of T) if T has no successors, or
 - b) $h(T) = u(T) + \max\{h(T') : T' \text{ a successor of } T\}$ if all successors of T have a height assigned.
- (2) Assign a depth, $d(T)$, to each task T in exactly the same way as $h(T)$ was defined, but substituting "predecessor" for "successor" in all places in the definition.
- (3) Let $e = \min\{u(T) : \text{all } T \text{ in } J\}$.
- (4) For each natural number j , let $Q_j = \{T : h(T) - u(T) \geq j \times e\}$.
- (5) Likewise define $R_j = \{T : d(T) - u(T) \geq j \times e\}$.
- (6) Let $CPT = \text{critical path time} = \max\{h(T) : \text{all } T \text{ in } J\}$
 $= \max\{d(T) : \text{all } T \text{ in } J\}$.
- (7) For any set C of tasks, let $TPT(C) = \text{total processing time of } C = \sum_{T \in C} u(T)$.
- (8) Define

$$\text{Size}(J) = \max\left\{ CPT, \max\{j \times e + TPT(Q_j)/m : 0 \leq j \leq \lceil CPT/e \rceil\}, \right. \\ \left. \max\{j \times e + TPT(R[j])/m : 0 \leq j \leq \lceil CPT/e \rceil\} \right\}.$$

The quantity $\text{Size}(J)$ so defined is readily calculated in time $O(n)$, where n is the number of tasks, and so can be efficiently used for setting job priorities in a scheduling algorithm. $\text{Size}(J)$ gives a reasonable lower bound on the time to complete J under any schedule of J on m processors; hence selecting first the job J with smallest $\text{Size}(J)$ would be a reasonable strategy. In order to give more concrete support to this statement, we turn to Hu's original work [HU 61] in which he shows that, when all tasks are unit length and the precedence relation forms an in-tree, $\text{Size}(J)$ is the minimum time required to execute J .

PROPOSITION 2.1: If J is a collection of unit-time tasks, $T_1, T_2, \dots, T_n$, related by an in-tree precedence relation, then $h(T)$ = the number of tasks in the chain from T to the root, $Q_j = \{T: h(T) \geq j+1\}$, and $\lceil \text{Size}(J) \rceil = \max\{ \text{CPT}, \max\{ j + \lceil \#(Q_j)/m \rceil : 0 \leq j \leq \text{CPT} \} \}$ is the minimum time required to complete all the tasks.

PROOF: See Hu [HU61], pp. 844-847. Notice that the terms with R_j play no role when the DAG is an in-tree since the number of tasks at each depth, starting with the leaves at depth one, must form a non-increasing sequence. As long as there are m or more tasks at each depth, $j + \lceil \#(R_j)/m \rceil$ will form a non-increasing sequence as well, while once there are fewer than m leaves at level j , the same sequence will be non-decreasing. The maximum value of the sequence must therefore be at $j = 0$, giving TPT/m , or at $j = \text{CPT}$, giving CPT .

In cases involving tasks which are not unit time or the precedence relation is not an in-tree, $\text{Size}(J)$ is still a lower bound on the makespan of any schedule for J , since the jobs in Q_j (for any j) cannot be run on m processors in less time than $\text{TPT}(Q_j)/m$. After the last task T in Q_j has been run, its successor tasks cannot be completed in less time than any chain of them, the longest having length $h(T) - u(T) \geq j$ by the definition of Q_j . A symmetric argument shows that each $j + \text{TPT}(R_j)$ is also a lower bound. On the other hand, if the number of tasks at about the same height (or depth) varies widely, $\text{Size}(J)$ may underestimate the makespan of J . It is shown below that it never underestimates by more than a factor of two.

PROPOSITION 2.2: Let J be a job consisting of tasks related by an arbitrary precedence, and $S(J) = \max\{ \text{CPT}(J), \text{TPT}(J)/m \}$. If $F(J)$ represents the makespan (finish time) of J when run on m processors by any scheduler that never leaves

a processor idle unless necessary, then

$$1 \leq \frac{F(J)}{\text{Size}(J)} \leq \frac{F(J)}{S(J)} < 2 - \frac{1}{m}. \quad (14)$$

Moreover, for each m there exists a job with unit tasks for which the ratio $F(J)/\text{Size}(J)$ is arbitrarily close to $3/2 - 1/2m$.

PROOF: Let makespan $= x + y$, where x time units are spent with all m processors busy and y time units are spent with at least one idle processor. At any time at which there is an idle processor all of the available tasks—and hence all of the highest level tasks—must be running, so CPT is being reduced. Therefore $y \leq CPT$. If, on the other hand, all m processors are busy, then TPT is being reduced by m times the length of time; hence, $mx \leq TPT$ on this time interval. Finally, at least one processor is always busy, so whenever CPT is reduced by dt , so is TPT; hence, $mx + y \leq TPT$. Therefore the whole job takes time $x + y = (mx + my)/m \leq TPT/m + (m-1)CPT/m$. This gives

$$\begin{aligned} \frac{F(j)}{\text{Size}(J)} &\leq \frac{(m-1)CPT + TPT}{m \times \max\{CPT, TPT/m\}} \\ &\leq \frac{(m-1)CPT}{m \times \max\{CPT, TPT/m\}} \\ &\leq \frac{(m-1)}{m} + 1 \\ &\leq 2 - \frac{1}{m}. \end{aligned}$$

To prove the second part, consider, for any $m > 1$, the job consisting of $2r + n$ tasks with precedence relations

$$\begin{aligned} T_1 \rightarrow_I T_2 \rightarrow_I \cdots \rightarrow_I T_r; \\ T_r \rightarrow_I T_k \rightarrow_I T_{r+n+1}, \text{ for } k = r+1, r+2, \dots, r+n; \end{aligned}$$

and finally

$$T_{r+n+1} \rightarrow_I T_{r+n+2} \rightarrow_I \cdots \rightarrow_I T_{2r+n}.$$

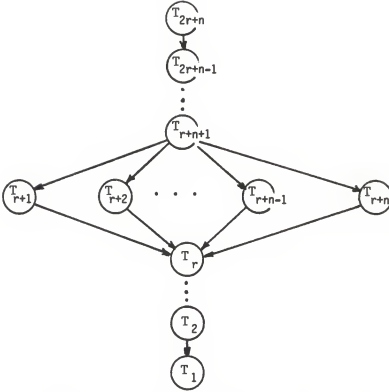


Figure 2.3. A Worst-Case Precedence Graph

Further suppose that $r = pm$ for some integer p and that $n = r(m-1) + m$. Then it is easily calculated that

$$CPT = 2r + 1,$$

$$TPT/m = (2r+n)/m = (r(m+1)+m)/m = r(1+1/m) + 1 < CPT,$$

and

$$\begin{aligned} j + \#(Q_j)/m &= j + (TPT-j)/m = j + (m(r+1) + r - j)/m \\ &= r + 1 + (j + r)/m, \text{ for } j = 1, 2, \dots, r. \end{aligned}$$

For $j < r$, $r + 1 + (j+r)/m \leq 2r + 1$, so $j + \#(Q_j)/m < CPT$. For $j \geq r$, $j + \#(Q_j)/m$ decreases even more and hence is always smaller than CPT . Finally, by symmetry, $j + \#(R_j)/m$ is always less than CPT , so that $Size(J) = CPT = 2r + 1$. To calculate $F(J)$ on the other hand, any schedule must run the first r tasks and the last r tasks sequentially; therefore no schedule can run

this job in less than

$$\begin{aligned}
 2r + \left\lceil n/m \right\rceil &= 2r + \left\lceil (r+1-r)/m \right\rceil \\
 &= 2r + 1 + \left\lceil (r-r)/m \right\rceil \\
 &= 2pm + 1 + (pm-p) \\
 &= 3pm - p + 1,
 \end{aligned}$$

giving

$$\frac{F(J)}{Size(J)} \leq \frac{p(3m-1)+1}{2pm+1} \frac{3m-1+1/p}{2m+1/p}.$$

This ratio is always less than $(3m-1)/2m = 3/2 - 1/2m$ and gets arbitrarily close to it as p (and hence r and n) gets large.

Whereas either CPT or TPT/ m by itself often makes reasonable estimates of the size of a job, there is no limit on how large the ratio of makespan to either of these measure can become as m tends to infinity. Nonetheless, as we have just seen, $Size(J)$ is never less than half the makespan. There are, however, even more clever measures for the "size" of J , but the more clever these measures become the more time is required to compute them. It is, after all, possible (in exponential time) to determine exactly how long the optimal makespan of any job is and use that as the size!

2.6. Conclusions

This chapter has explored a number of avenues leading to more efficient scheduling methods for running concurrent programs on general-purpose multiprocessor systems. In order to reduce the turnaround times of such jobs, a number of extensions of the traditional Shortest Job First algorithm have been presented, and a worst-case analysis of one of these--called Concurrent Shortest Job First (CSJF) or

just Shortest Job First—was made. This analysis indicates that the concurrent form of the algorithm can do no worse than its sequential counterpart. It also indicates that the ratio of the turnaround time of CSJF to the optimal turnaround time is bounded by m —the number of processors—and also by a multiple of the ratio of the longest to the shortest job.

As a result of such analysis, one is led to test CSJF against other scheduling methods to see if it produces the lowest average turnaround time. The results of a simulation experiment are presented and analyzed in Sections 2.3 and 2.4, comparing CSJF with several other related scheduling methods and with a random scheduler. These results indicate mostly negligible differences among the various schedulers, with the methods based on the Shortest Critical Path First strategy faring the best in the significant cases. A critique of the design of the simulation pointed out possible explanations for the small differences observed and indicated ways to improve future simulation experiments.

Finally, Section 2.5 developed a more sophisticated measure of the "size" of a concurrent program, or of a directed acyclic graph, in terms of both critical path and processing time of certain subsets of the program. This value, $\text{Size}(J)$, is shown to be a reasonably good lower bound for the optimal run time (makespan) of the job J —better than critical path time (CPT) or total processing time (TPT)—and hence a good candidate for a "Shortest Size First" scheduling algorithm.

CHAPTER III LOOSELY COUPLED SYSTEMS

Although it would seem that allowing multiple processors to share the same central memory store would make their cooperative efforts simpler and faster, such memory sharing creates a quantity of difficulties that grows rapidly with increasing numbers of cooperating CPUs. The major problems here are first to provide the necessary hardware for multiple direct access to the memory, and second to assure, by whatever means, fast and conflict-free access to the memory for all processors. Solutions to these problems become expensive and complex, but experimentation continues in many directions by many organizations. The alternative is to provide a separate memory for each CPU (or small group of CPUs), forcing communication between two CPUs to be done by some kind of message passing. These are the "loosely coupled" systems. The obvious disadvantage of overhead associated with the transmission of messages is often outweighed by the savings in hardware complexity and by increased flexibility. Naturally, the type of concurrent problem solving appropriate on a tightly coupled system might be inappropriate on a loosely coupled one: fine-grained concurrency such as sharing the evaluation of parts of an expression can create speed-up with shared memory, but the communication delays would make this sharing useless in a loosely coupled system.

The scheduling strategies which are appropriate for a tightly coupled system may also not be adequate for one which is loosely coupled. Moreover, even if all processors can run tasks at the same rate, the type of communication links among the processors may dictate that certain combinations of task-processor assignments are more efficient than others. The hypercube architecture, for example, has each processor in direct communication with some neighboring processors, but messages to

other processors must be forwarded by one or more intermediate processors. This, of course, means that scheduling successive tasks on the same processor or near neighbors should produce less communication overhead and shorter makespans than if these tasks were spread over processors separated by more intermediate links. This is a different kind of difficulty than those encountered in the "classical" scheduling problems discussed in the foregoing chapters. It, therefore, leads to a scheduling model considerably different from the one in Chapter II and also considerably different from that studied by former scheduling theory researchers. In industrial scheduling environments, setup times and machine differences have been considered, but apparently not delays which depend on which machines were used to process predecessor tasks.

This chapter also looks at a different performance measure: to minimize the total time—or makespan—required to complete a given set of precedence-related tasks on a loosely coupled system. The first section deals with the more detailed assumptions that are made on the communications between processes in order to develop a tractable scheduling model. Section 3.2. then presents the main result, which is an optimal scheduling algorithm for a particular case of the new model. The final section discusses some extensions of the basic algorithm which are significant in their own right.

3.1. Scheduling and Communication

In discussions of the classical scheduling problems without communication overhead, the time required to make the scheduling decisions themselves is usually ignored, even in the dynamic case. This assumption of no scheduling overhead is also made in this section; however, further assumptions as to the nature of the communication overhead are also necessary now. As seen in the last section, the type of

architecture affects substantially the appropriate assumptions. Nonetheless, there are a few basic requirements that are imposed throughout the remainder of this chapter:

- (1) All communications consist of a number of "message units." The number of messages, $m(T, T')$, which must be sent from one task T to an immediate successor task T' is a fixed integer > 0 , independent of the processors on which T and T' are scheduled.
- (2) The time, $d(P, P')$, required to send one message unit from processor P to processor P' in the absence of contention is a system constant depending only on P and P' . Moreover, $d(P, P') = 0$ if and only if $P = P'$. The time required for the channel protocol to schedule message transmission is constant and forms part of the time $d(P, P')$.
- (3) Communication protocols are collision-free, so that no messages are lost and all messages are sent in a finite amount of time.
- (4) In the presence of contention for a particular channel, the time required to transmit a collection of message units is just the sum of their transmission times plus the transmission times of any messages for which they must wait.
- (5) The channel processors are independent of the task processors, implying that all processors may be running tasks at the same time that communication is taking place among the processors.
- (6) All messages are sent at the time of completion of the originating task, and the receiving task cannot begin until all messages are received from all preceding tasks.

It is instructive to look at the effect of assuming—or not assuming—each one of these restrictions. The first two make the possible set of communication delays a discrete, finite set: without (1), messages could be of arbitrary lengths, while without (2), the amount of time necessary to send the same message might vary from one moment to another. Of course, if there is contention for the communication channel,

then one or more messages may have to wait for the transmission of others, thereby changing the time required for the messages to be received. Notwithstanding this complication, assumptions (3) and (4) guarantee that this time will always be finite and predictable, allowing for deterministic scheduling policies. If one of these restrictions were not to hold, then the scheduling problem would be non-deterministic, as there would be no way to predict exact communication delays. Assumption (5) simply says that communications are not to be treated as extra tasks to be scheduled and executed on the given processors: this corresponds to a hardware assumption of "intelligent" I/O processors. The final restriction, assumption (6), is the least critical. While (5) makes it clear that there can be "overlapping" of communication times and execution times, (6) says that communication between T and T' cannot overlap either T or T' . This final assumption could be relaxed and still produce a deterministic scheduling problem.

Each of the foregoing assumptions on the nature of the communications corresponds to certain assumptions on the nature of the hardware and communications software of the system. In the case of a system such as the hypercube, which is not fully connected, Assumption (2) must be modified to apply only to two processors which can communicate directly with one another; beyond that, the time required to send a message from P to P' must be calculated on the basis of the route chosen and the contention encountered on each leg of the communication route.

A large number of static, deterministic scheduling problems can now be precisely stated under these assumptions. This is done in Hwang et. al [HWAN86], which presents several different problems of this kind and indicates that there are thousands more depending on the selection of the architecture and the traditional parameters of the system. It would be of great interest to begin to draw the boundaries between the NP-hard problems and the polynomial-time problems in this new problem space.

Perhaps the most encompassing attempt to attack the general problem of minimizing the makespan of a schedule in the presence a general DAG and communication delays is found in the Ph.D. dissertation of J.-J. Hwang [HWAN87] in which he presents an intelligent heuristic algorithm called Earliest Task First (ETF). ETF is a "greedy" strategy which, at any moment that a processor becomes free, attempts to schedule some task as early as possible on that processor. Although the strategy is not optimal, Hwang establishes a worst case bound on its performance which is cited in Proposition 3.1.

PROPOSITION 3.1: Given a set of tasks with a general precedence relation and given a loosely coupled system of m identical processors satisfying the conditions (1) to (6) above, let M_{ETF} be the makespan of a schedule produced by ETF and M_{opt} be the optimal makespan. Then

$$M_{ETF} \leq (2 - \frac{1}{m}) \times M_{opt} + MaxChainComm, \quad (1)$$

where $MaxChainComm$ is the maximum sum of the form $\sum MaxDelay \times m(i,j)$, the sum taken along any chain of tasks in the system. $MaxDelay$ is the maximum of all communication parameters $d(P, P')$ taken over all pairs of processors.

3.2. An Algorithm for Precedence Trees

In order to obtain an optimal scheduling algorithm in the case of non-negligible communication times, it is necessary to make even greater restrictions on the problem than those imposed in Section 3.1. To begin with, the corresponding problem without communication delays must be solvable in polynomial time. As indicated at the outset of the chapter, this discussion focuses only on the problem of minimizing the makespan of a set of tasks. Since there would be no communication if the tasks were independent, it is assumed that there is a precedence relation among them,

representable by a DAG as always. Since in the classical case, minimizing the makespan is NP-hard even for unit execution times (UET) [ULLM75], further restrictions must be imposed. For two processors and UET, an algorithm by Fuji, Kasami, and Ninomiya is optimal [FUJI69], as is the better-known algorithm of Coffman and Graham [COFF72]. In this case, allowing two different possible execution times again makes the problem intractable [ULLM75]. On the other hand, restricting the precedence relation to an opposing forest (each connected component of the DAG is a tree) and restricting the number of processors to any fixed constant again produces a polynomial time scheduling problem [GARE83]. If m is an arbitrary parameter of the problem, polynomial time solutions are still possible if the DAG is further restricted to be a forest of in-trees (all out-degrees = 1) [HU 61] or a forest of out-trees (all in-degrees = 1) [BRUN82]. The general case of arbitrary m and an opposing forest remains open [DOLE85] in the classical case.

The obvious problems to examine in the case of significant communication overhead, therefore, are those with UET and either two processors or with many processors and in-tree or out-tree forests. An algorithm which the author conjectures is optimal for in-tree forests and arbitrary m is presented in the next section: here different assumptions are introduced. Whereas in the classical case, if there are more processors than tasks then the scheduling problem becomes trivial, in the new situation, the problem of allocation of tasks to processors still remains difficult. Consider the example of the DAG given in Figure 3.1, where each task is assumed to have unit execution time. With no communication overhead, this can be scheduled on two processors, as in Figure 3.2 (a), to execute with an optimal makespan of three. If communication times of 0.5 are supposed between any combination of tasks and processors, then wherever T_4 is scheduled, it will have to start at least 0.5 time unit later than either T_2 or T_3 , so Figure 3.2 (b) shows an optimal schedule in time 3.5. If the delays due to the communications varied between different tasks and

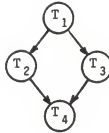


Figure 3.1. A DAG of Unit-Time Tasks

processors, then the assignment to processors would be even more critical and the makespan could be even longer. However, if the delays become greater than one, then the schedule of length four on one processor, shown in Figure 3.2 (c), becomes optimal.

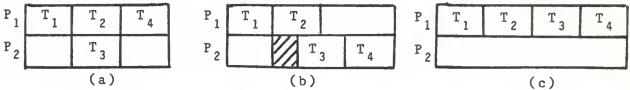


Figure 3.2. Three Optimal Schedules

Still working with the same DAG of Figure 3.1, another significant idea emerges if one considers the case where communication time is, say, 1.5 between T_1 and its immediate successors but only 0.5 between these successors and T_4 . Then it would appear that again Figure 3.2 (c) would be an optimal schedule since placing T_2 and T_3 on different processors would make one of them wait until time 2.5 to start and

produce a schedule of length 4.5. This can be avoided, however, by running T_1 on both P_1 and P_2 ! This extra use of memory space and processor time allows the creation of the optimal schedule of length 3.5 shown in Figure 3.3. (See [PAPA87].)

P_1	T_1	T_2		T_4
P_2	T_1	T_3		

Figure 3.3. An Optimal Schedule with Task Duplication

It is clear, therefore, that assuming "sufficient" processors to run all available tasks at any moment does not trivialize the scheduling problem with communication overhead the way it does the classical problem. In fact, without further assumptions the problem remains quite complex. The algorithm presented below is shown to be optimal under the additional restriction that the communication delays are not longer than the task execution times; notwithstanding, it also works without the usual UET assumption.

Assume that there are given n tasks, $T_1, T_2, \dots, T_n$ satisfying a precedence relation and m identical processors, $P_1, P_2, \dots, P_m$. Assume that the processors are loosely coupled, so that the communication time between them is not negligible with respect to the processing times of the tasks. Let u_i represent the processing time of task T_i , and assume that the six communication assumptions of Section 3.2 hold in this system. Finally, the following restrictions should be assumed:

- In-Forest Precedence: The precedence DAG of the tasks is in the form of a forest of in-trees (all nodes with outdegree ≤ 1).
- Sufficient Processors: $m > n$. (In fact, $m \geq$ the number of leaves of the forest is a sufficient condition.)
- Short Communication Delays: The time required for any task to communicate its results to an immediate successor is less or equal to $\min\{u_i; 1 \leq i \leq n\}$ in all cases and is equal to 0 if scheduled on the same processor.
- Identical Links: The time $d(P, P')$ required to send a message unit from P to P' is constant, independent of the processors. (It is, of course, 0 if $P = P'$.)
- Fully Connected: All processors can communicate directly with all others without contention. Thus any number of processors may communicate with any others simultaneously.

The following algorithm determines an assignment of the n tasks to m processors (for sufficiently large m) in such a way as to minimize the makespan of running all the tasks. It uses the scheduling strategy of joining each task with that predecessor which would otherwise cause the longest delay: for that reason it is named Join Latest Predecessor.

Algorithm 3.1. JLP (join latest predecessor)

Input: Tasks $1, 2, \dots, n$, with processing times $u_1, u_2, \dots, u_n$; precedence relation $\rightarrow$ such that for any $i, i \rightarrow_I j$ for at most one j ; communication delays $c[i, j]$ for each $i \rightarrow_I j$ such that $c[i, j] \leq u_k$ for all i, j, k . Further assume that the tasks are numbered such that $i \rightarrow j$ implies that $i < j$ ("topological order").

Output: For each $i \leq n$, 3 numbers: $P(i)$, indicating the processor on which task i should be scheduled; $S(i)$ and $F(i)$ indicating the start time and finish time, respectively, of task i on processor $P(i)$.

BEGIN

1. FOR $j = 1$ TO n

BEGIN

IF j is a leaf (no predecessors)

1.1. THEN Set $P(j) = j$, $S(j) = 0$, $F(j) = u_j$.

{ j is now "scheduled." }

1.2. ELSE

BEGIN

1.2.1. Find an immediate predecessor k such that $F(k) + c[k, j]$ is maximum for all immediate predecessors of j .

1.2.2. Set $P(j) = P(k)$.

{ Assures that j need not be delayed by $c[k, j]$. }

1.2.3. Set $S(j) =$

$\max\{ F(k), \max\{ F(i) + c[i, j] : i \rightarrow_I j \text{ and } i \neq k \} \}$.

{ j will start when k finishes or when the latest communication arrives from its other immediate predecessors. }

1.2.4. Set $F(j) = S(j) + u_j$.

{ j is now "scheduled." }

END ELSE

END FOR

END JLP.

It is now necessary to establish that this algorithm does, in fact, produce the best possible assignment of tasks to processors in the sense that the schedule produced minimizes the makespan. This is the content of the next theorem.

THEOREM 3.1: The schedule produced by the JLP algorithm yields the minimum possible makespan under the five hypotheses set out before the algorithm.

PROOF: That the schedule is feasible is clear from Step 1.2.3, which starts the job j only after all predecessors have finished and all messages have had time to arrive. Step 1.1, of course, depends upon having sufficient processors, but note that the algorithm produces a feasible schedule even without the Short Communication Delays assumption.

Now suppose, for the sake of contradiction, that there is a better schedule than that of JLP which gives finish times $F'(i)$, for $i = 1, 2, \dots, n$. Then there must be a j such that $F'(j) < F(j)$ and $F'(i) \geq F(i)$ for $i < j$. Since all leaves have $F(i) = u_i$, such a j cannot be a leaf. Let k be the predecessor of j found in Step 1.2.1 of the algorithm. $k < j$ due to the topological order, so by the choice of j , $F'(k) \geq F(k)$; hence j cannot start before $F(k) + c[j, k] \geq S(j)$ unless j is run on the same processor as k . Similarly, if i is any other predecessor of j , then $F'(i) \geq F(i)$. Therefore, if j runs on the same processor as k , it cannot start before the time $S(j)$ given in Step 1.2.3 unless some other immediate predecessor, say r , runs on the same processor as k and j . In this case, if $F'(r) < F'(k)$, then $F'(k) \geq F'(r) + u_k \geq F'(r) + c[r, j]$ by the short communications assumption. But $F'(r) + c[r, j] \geq F(r) + c[r, j]$, so scheduling r on the same processor as j does not allow j to start any earlier than $F'(k) \geq F(r) + c[r, j]$, which is no improvement over JLP. Symmetrically, if r runs on the same processor as j and k and if $F'(r) > F'(k)$, then it follows that $F'(r) \geq F(k) + c[k, j]$, which again offers no

improvement over JLP. Therefore $S'(j) \geq S(j)$, a contradiction. It follows that no other schedule produces a shorter makespan than JLP. $\square$

An example of a JLP schedules appears in Figure 3.4.

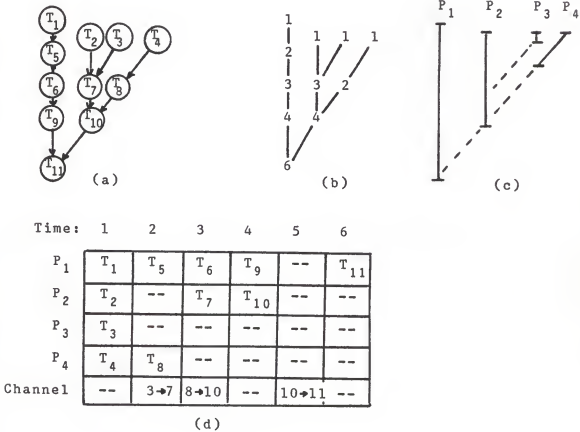


Figure 3.4. A JLP Schedule. (a) UET, UCT Tasks (b) Finish Times (c) Processor Assignments (d) JLP Schedule

THEOREM 3.2: The time complexity of the JLP algorithm is $O(n)$; that is, the time required to produce the schedule is linear in the number of tasks to be scheduled. (This assumes that the precedence relation is given in terms of immediate successors as shown in the algorithm. If the tasks are not in topological order, the algorithm requires minor revision, but Theorem 3.2 remains true.)

PROOF: JLP can do no better than $O(n)$, since the main loop, Step 1, executes n times. This bound, $O(n)$, will be achieved provided that the total number of steps required, during all n iterations, to find the predecessor k in Step 1.2.1 and to calculate the value of $S(j)$ in Step 1.2.3 is $O(n)$. In order to do this, a list of immediate predecessors must be initialized (in $O(n)$ time) for each task during the input phase. It is easy to look through the list of predecessors of a task j once only and calculate the maximum and second largest values required in Steps 1.2.1 and 1.2.3. Since the DAG is a forest of in-trees, each task appears at most once as a predecessor of some other task, guaranteeing that over the n iterations of the main loop only $O(n)$ steps go into these calculations. $\square$

3.3. Extensions

There are a number of extensions that can be made to the JLP algorithm under different relaxations or changes in the hypotheses. An obvious place to start is to look at a forest of out-trees, keeping the other hypotheses of Section 3.2 the same. In this case, however, if task duplication is allowed, as illustrated in Section 3.2, Figure 3.3, an essentially trivial algorithm always produces an optimal schedule with no communication delay at all. The algorithm would, for each leaf j , schedule all the tasks on the unique path from the root to j on processor j . Since, in an out-tree, each task has a unique predecessor, the tasks scheduled on any one processor have all their predecessors scheduled on the same processor and hence there is never any need for message passing. This procedure requires multiple copies of most tasks—a space complexity of $O(n^2)$ —but produces a makespan equal to the critical path time of the DAG, which is always a lower bound on the makespan of any schedule. In order to implement the algorithm, it is only necessary to obtain, for each task, the list of its immediate successors and the count of the number of leaves which are successors of

each task. Since this count, for any task, is equal to the sum of the counts of each of its successors, the count can be calculated, starting with the leaves (highest numbered tasks), in $O(n)$ time. Once the count of leaves is obtained, call it $C(j)$ for each j , the scheduler simply schedules, starting with the root, $C(j)$ copies of j on $C(j)$ of the processors running j 's unique predecessor. This also takes $O(n)$ time.

Slightly more interesting is the same problem of a forest of out-trees when duplications of tasks are prohibited. Some communication delays are now unavoidable, but due to the assumptions of Sufficient Processors and Identical Links, it is possible to turn the DAG upside down and use JLP. More specifically, define the dual DAG by replacing $i \rightarrow j$ by $n-j+1 \rightarrow n-i+1$ and setting $c[n-j+1, n-i+1] = c[i, j]$ for every precedence-related pair. If the original DAG was an out-forest, an in-forest is obtained, and vice versa. It is clear that the dual of the dual is again the original DAG and communication times. Now, if JLP is applied to the dual of the given out-tree, obtaining starting times $S(j)$, finish times $F(j)$, and a makespan of M , then an optimal schedule for the original problem is obtained by setting

$$\begin{aligned} S'(j) &= M - F(n-j+1), \\ F'(j) &= M - S(n-j+1), \end{aligned}$$

and assigning all tasks to the same processors assigned by JLP. If this algorithm is called JLP', then the following result holds.

THEOREM 3.3: Given the same assumptions used with the JLP algorithm except that the precedence relation gives a DAG which is an opposing forest (a disjoint union of in-trees and out-trees), and if duplicate copies of tasks are not executed on more than one processor, then scheduling the in-forest on one set of processors with JLP and the out-forest on a disjoint set of processors with JLP' produces an optimal schedule with respect to the makespan.

PROOF: Given sufficient processors, running the in-trees and out-trees on disjoint sets of processors creates a makespan which is the maximum of the makespans of the two disjoint sets. If each is optimal, then the larger is the optimal makespan for the whole opposing forest. It has already been established that JLP is optimal, so it remains to be shown that JLP' is optimal. But JLP' on an out-tree produces a schedule the same length as JLP does on the dual in-tree. If there were a shorter schedule for the out-tree, taking the dual schedule for the dual in-tree would produce a shorter schedule for the in-tree: a contradiction to the optimality of JLP. The only problem, therefore, is the feasibility of the JLP' schedule. As mentioned before, this is a consequence of the Sufficient Processors and Identical Links assumptions: sending a message from P to P' takes the same time as sending a message from P' to P . This means that when Step 1.2.3 of JLP is performed, guaranteeing that j does not start before all its predecessors' messages have been received, it also guarantees that all of the successors of $n - j + 1$ in the dual tree do not start before all have received their messages from their sole predecessor. Hence the feasibility of the schedule is assured. (If duplications are allowed in scheduling the out-trees, this symmetry is destroyed since it is not possible to run task j just because every required message has been received by some copy of j .) $\square$

Having extended the JLP algorithm to handle all opposing forests under the assumption of Sufficient Processors and Short Communication Delays, it is natural next to return to in-trees (or forests of in-trees) and ask if JLP will work if one or the other of these assumptions is removed. It is evident that JLP will continue to produce feasible schedules with any length communication delays, but it does lose its optimality. For example, if communication delays, for example, are as long as the total processing time of all the tasks combined, the best thing to do is run all tasks

on a single processor; JLP, however, always insists in starting out with all leaves on different processors.

Dropping the assumption of Sufficient Processors, in the other direction, leads JLP into trouble immediately, since Step 1.1 cannot always be carried out. There is, however, a value to JLP in this case because it always provides a lower bound on the time required to execute any task in the in-tree.

LEMMA 3.1: Given a DAG of n tasks with communication delays and m identical processors satisfying all the five hypotheses of the JLP algorithm, the time $S(j)$ produced by JLP is the earliest time that j can be started by any scheduling algorithm. If the hypotheses of Sufficient Processors and In-Forest Precedence are dropped, $S(j)$ remains a lower bound on the starting time of j in any schedule (unless duplicate copies of tasks are allowed).

PROOF: That each task is optimally scheduled by JLP is what was actually demonstrated in the proof of Theorem 3.1. That with a limited number of processors no shorter schedule is possible than with an unlimited number is obvious (Graham's timing anomalies [GRAH69] notwithstanding). Without the assumption of unique successors, JLP will frequently schedule successors of a task on the same processor at the same (or overlapping) time. The schedules so obtained are not feasible, but the times, $S(j)$, calculated by the algorithm can still only be better than those corresponding to a feasible schedule. $\square$

Hu is credited with the first formal proof that the Highest Level First (HLF) scheduling policy could produce optimal schedules [HU61]. HLF always schedules first one of the available tasks of highest "level" in the DAG. "Level" is defined the same as "height" in Section 2.5. A quarter century ago, Hu showed that this strategy works for minimizing the makespan on any number of processors when the tasks are unit execution time (UET) and the precedence is an in-tree [HU61]. Consider now

now the case of a loosely coupled system of m processors and an in-tree DAG of n UET tasks with "unit communication times" (UCT): all $c[i, j] = 1$ unless i and j are scheduled on the same processor. HLF is a tempting strategy here, but fails because it ignores the added "height" caused by the communication. Figure 3.5 shows a case in which, on three processors, HLF may obtain a schedule of length six, while the optimal schedule is of length five.

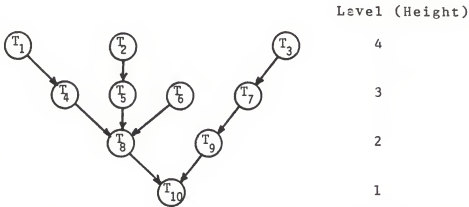


Figure 3.5. A UET, UCT Problem for which HLF is suboptimal

Time:	1	2	3	4	5	6
P ₁	T ₁	T ₄	--	T ₈	--	T ₁₀
P ₂	T ₂	T ₅	--	--	--	--
P ₃	T ₃	T ₆	T ₇	T ₉	--	--
Channel	--	--	5-8 6-8	--	9-10	--

Figure 3.6. Suboptimal HLF Schedule for Figure 3.5

JLP ferrets out the unavoidable communication delays and allows an extended definition of height which looks very promising as the basis of a new HLF algorithm for loosely coupled systems. In the above example, it makes task 4 lower than tasks

5, 6, and 7, forcing a HLF strategy into the optimal strategy of scheduling task 7 rather than 4 in the second time slot and allowing 9 and 10 to start one time unit earlier. The algorithm that follows, called Extended JLP, or EJLP, presents this formally.

Algorithm 3.2. EJLP (extended join latest predecessor)

Input: Tasks 1, 2, ..., n , (with unit processing times); precedence relation $\rightarrow_I$ such that for any i , $i \rightarrow_I j$ for at most one j ; Further assume that the tasks are numbered such that $i \rightarrow j$ implies that $i < j$ ("topological order"); an integer m indicating the number of processors.

Output: For each $i \leq n$, 2 numbers: $P'(i) \leq m$, indicating the processor on which task i should be scheduled; $S(i)$ indicating the start time of task i on processor $P'(i)$.

BEGIN

1. Execute JLP to assign a "processor number," $P(i)$, to each task i .
 $\{ P(i) \text{ may be } > m. \text{ It is only used to determine height. } \}$
2. FOR $j = n$ DOWN TO 1 { Assign a height, h , to each task. }

BEGIN

IF j has no successors

2.1. THEN Set $h(j) = 1$

2.2. ELSE { Let k be the unique successor of j . }

IF $P(j) = P(k)$

2.2.1. THEN Set $h(j) = h(k) + 1$

2.2.2. ELSE Set $h(j) = h(k) + 2$

END FOR

3. Execute HLF using the values $h(j)$ to define the height, or level. A task j is only available for scheduling at time t , however, if all its predecessors are scheduled for time $\leq t-1$ and at most one of its predecessors is scheduled to start at time $t-1$. $S(j)$ is the start time assigned to j by this HLF algorithm. $P'(j) \leq m$ is arbitrary, except that
- (a) if $S(j) = S(i)$, then $P'(j) \neq P'(i)$, and
 - (b) if exactly one predecessor, k , of j is scheduled at time $S(j)-1$, then $P'(j)$ must be equal to $P'(k)$.

END EJLP.

LEMMA 3.2: The EJLP Algorithm produces a feasible schedule for an in-tree DAG under the UET and UCT assumptions.

PROOF: Step 1 can be carried out since all assumptions for JLP except Sufficient Processors are in effect and the numbers $P(j)$ are not to be interpreted as actual assignments to processors. Step 2 makes sense because by taking the tasks in reverse order, the topological ordering guarantees that if k is the immediate successor of j , then $k > j$, so $h(k)$ is assigned before considering task j . Finally, HLF assigns ready tasks to available processors and presents no problem when there is no communication overhead. Under the assumptions for EJLP, all tasks and all communication times are of length one; therefore each task will become ready—all predecessors scheduled and all messages received—at some integer time $t = 0, 1, 2, \dots$. If all but one of the predecessors of a task j are scheduled to start before time $t-1$, then by time t , all messages to j except those of the latest predecessor, say i , have been received. By scheduling j on the same processor as i , no messages need be sent from i to j and hence j can start at time t .

If, on the other hand, two predecessors of j are scheduled to start at time $t-1$, then no matter to what processors they are assigned, j will have to wait for messages

from one of them and hence cannot start until time $t+1$. The conditions presented in Step 3 precisely assure that tasks are not scheduled before it is feasible to do so. Since there is a maximum of one successor for each task, condition (b) of Step 3 can always be met. $\square$

CONJECTURE: For $m < 6$ processors, EJLP always produces a schedule with minimum makespan for a set of UET, UCT tasks satisfying the In-Forest Precedence, Identical Links, and Fully Connected hypotheses.

The unusual number $m = 6$ features in this conjecture because it can be shown that for $m < 6$ that EJLP has to reduce the "height" calculated by the algorithm in such a way as to even out the heights of the highest remaining tasks. In the example of Figure 3.7, however, one valid EJLP schedule takes the rightmost six leaves at time zero, the leftmost six leaves at time one, the rightmost six new leaves at time two, and all available tasks from then on, producing a schedule of length six. In this case, the two leftmost subtrees are assigned heights of three and four, respectively, and the EJLP schedule indicated reduces these to one and three at time $t = 1$ and then to heights zero and two at time $t = 3$. The schedule produced is of length six, while an optimal schedule would be of length five.

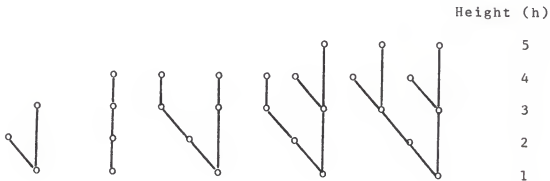


Figure 3.7. An In-Forest Where EJLP Fails with $m = 6$

It is important to observe in this example, as in the example of Figure 3.6, that the algorithms being studied could produce optimal schedules in the given cases. An algorithm is considered optimal, however, only if any schedule which follows the rules of the algorithm must be optimal. For the DAG of Figure 3.7, EJLP could also have chosen the three tasks of height five and the three leftmost tasks of height four at time zero and proceeded to produce an optimal schedule of length five. It is tempting to conjecture that EJLP would be optimal for all m if modified to pick, among tasks of the same height, in such a way as to minimize the number of tasks left "blocked." (A task is blocked if all its predecessors have been scheduled but it must still wait for a message. Blockage occurs only when the last two or more predecessors were scheduled at the last time interval. The effect of blockage is reflected in Step 2.2.2 of EJLP.)

The foregoing paragraphs have considered what happens when the assumptions of the JLP algorithm are relaxed by allowing longer communication times or by restricting the number of processors. As a final investigation, the Short Communication Times and Sufficient Processors hypotheses are reinstated, but all restrictions on the DAG are dropped. Recalling Proposition 3.1 presented in Section 3.1, assuming m is arbitrarily large and communication delays are shorter than task processing times, the worst case bound for applying the ETF strategy becomes

$$M_{ETF} \leq 2M_{opt} + CPT, \quad (2)$$

where, as usual, CPT = Critical Path Time is the longest total processing time along any path in the DAG. Since the makespan is never less than CPT, this could be weakened slightly and written simply as

$$M_{ETF} \leq 3M_{opt}. \quad (3)$$

Surprisingly, it is possible to produce an optimal scheduling algorithm for the case of Sufficient Processors and Short Processing Times provided duplicate executions of the

same tasks are once again allowed. The optimal strategy is also an extension of the basic JLP algorithm. It is based on the observation that any DAG, through duplication of tasks with out-degree greater than one, can be turned into an in-tree whose execution will produce the same result as the execution of the original DAG. It should be warned, however, that whereas such duplication makes perfectly good sense in computer science, as a model for an assembly procedure or industrial planning it could be complete nonsense. An informal description of an algorithm (JLP/D) embodying this idea appears below. The algorithm is based on a suggestion by an anonymous reviewer of a paper on the JLP algorithm submitted for publication by the author, J.-J. Hwang, and Y.-C. Chow.

Algorithm 3.3. JLP/D (JLP with task duplication)

Input: Tasks $1, 2, \dots, n$, with processing times $u_1, u_2, \dots, u_n$; arbitrary precedence relation $\rightarrow_I$; Communication delays $c[i, j]$ for each $i \rightarrow_I j$ such that $c[i, j] \leq u_k$ for all i, j, k . Further assume that the tasks are numbered such that $i \rightarrow_I j$ implies that $i < j$ ("topological order").

Output: For each $i \leq n$, 3 numbers: $P(i)$, indicating the processor on which task i should be scheduled; $S(i)$ and $F(i)$ indicating the start time and finish time, respectively, of task i on processor $P(i)$.

BEGIN

Execute JLP to assign processor numbers, $P(i)$, start times, $S(i)$, and finish times, $F(i)$ to each task i , but with the following modification: Whenever, in the j -th iteration of the main loop, JLP schedules task j in such a way that $P(j) = P(i)$ and the intervals $(S(j), F(j))$ and $(S(i), F(i))$ overlap for some $i < j$, copy the part of the schedule consisting of all the predecessors of j onto a disjoint set of processors. If JLP scheduled j on the same processor as its immediate predecessor k , then reassign $P(j) = P(k')$, where k' is the copy just

created of k . (In succeeding iterations of the main loop, it is not necessary to consider the copies created in the calculations performed in Steps 1.2.1 and 1.2.3.)

END.

THEOREM 3.4: JLP/D is optimal with respect to makespan for arbitrary DAGs run on loosely coupled systems and satisfying the Sufficient Processors and Short Communication Times hypotheses.

PROOF: As a consequence of Lemma 3.1, JLP/D produces an optimal schedule provided that it is feasible, since it starts all tasks at the times given by JLP. (It is crucial here that Steps 1.2.1 and 1.2.3 of the JLP algorithm do not depend on the previous assignments of tasks to processors—only on the $S(i)$ and $F(i)$. Therefore, the copying and reassigning of processors will not change the value calculated for $S(j)$, only the processor assigned to it.) To see that JLP/D creates a feasible schedule, refer to the proof of feasibility in the case of a forest of in-trees to see that, even for arbitrary precedence relations, the starting times, $S(i)$ are late enough to assure that all predecessors (and all their copies) have finished execution and enough time has passed for all messages to have arrived at the chosen processor. Moreover, the modification to JLP precisely checks for any problem due to two tasks being scheduled simultaneously on the same processor and changes processors to avoid the conflict. Since all predecessors of the task so moved are also copied, a simple induction proof establishes that the new schedule produced remains feasible. $\square$

CHAPTER IV OTHER MULTIPROCESSOR SCHEDULING PROBLEMS

In the course of this work, a large number of different scheduling problems have been discussed. Despite their differences, however, they all share many basic characteristics. Most of ground rules were presented in Chapter I or in the discussion of communication in Chapter III; nonetheless, some were tacitly assumed. This closing chapter takes a look at some computer systems in which different assumptions are appropriate and also examines some different kinds of scheduling problems with the intention of providing both contrast for the foregoing work as well as future avenues of extension.

4.1. More MIMD Scheduling Problems

Several performance measures were introduced and briefly discussed in Section 1.1 in order to emphasize that meaningful scheduling must always be related to the achievement of some measurable goal. Although improving one or more of the performance measures used to evaluate a system may be the long range goal, scheduling methods are usually directed at optimizing some more immediate values. An industrial organization, for example, may wish to lower the dollar value of its inventory of raw materials through more careful scheduling of its manufacturing process. Rather than talking about minimizing inventory, however, the relationship between turnaround times and inventory can be exploited: longer average turnaround time means higher average number of jobs waiting to be completed and therefore greater amounts of the necessary raw materials must be available. In a data processing center that wishes to maximize its profit, there may be more income for finishing

more jobs per day and also penalties for jobs finished after established deadlines. In this case, scheduling methods minimizing the number of late jobs or minimizing the makespan of a collection of jobs may be selected.

Chapter III gave much attention to minimizing the average turnaround time, one of the most common objectives of scheduling problems in the literature. Notwithstanding, it is recognized that policies optimizing average turnaround time also prejudice long jobs. If user satisfaction is linked to getting jobs done as soon as possible, lowering average turnaround time should mean that, on average, customer satisfaction is increased. The difficulty is that some few customers may become very dissatisfied at the same time. It may, therefore, be found better to try to minimize the maximum turnaround time, causing some small displeasure for customers with longer waiting times but assuring all of reasonable time to completion. Even more "fair" to customers would be to try to minimize the variance of the turnaround times without allowing the average turnaround time to increase too much. Policies such as Shortest Job First tend to increase variance rather than minimize it; hence, practical computer schedulers concerned with customer satisfaction use some form of modified SJF which raises priorities on jobs that have had to wait a long time for service.

Of course, many more esoteric objective functions have been used. For example, minimizing root mean square tardiness tends to avoid very tardy jobs more than if the objective is just to minimize the number of tardy jobs or the average tardiness. Many scheduling problems are posed in terms of optimizing some sort of weighted average, counting the completion of more "important" jobs more heavily than others. In general, the choice of objective function depends on many factors and can be difficult; in the end, however, this choice is often dictated by the need for simplicity in producing a tractable problem. Most research concentrates on a small number of possible objectives, largely because other objectives present far more difficult problems for analysis.

Besides changing the objective function, several kinds of assumptions can be made on the processors. In their computerized summary of scheduling results referred to already in this work, Lageweg et al. [LAGE81b] include the cases in which processors are equivalent but work at different speeds, in which processors are completely unrelated in their capabilities, and in which processors are of k different types corresponding to k different operations which must be performed on each job. Many multiple processor systems (MPS) contain a variety of processors or processors which cannot work independently of one another. Additionally, intelligent I/O channels are processors dedicated to specialized activities. Careful modeling of such systems requires different assumptions on processors as well as on job structure. In Section 4.2 of this chapter, more will be said about specialized computer architectures and their scheduling problems.

It is important to keep in mind that the added complication of interprocessor communication overhead in loosely coupled systems places these scheduling problems completely outside of the traditional classification schemes. This extra aspect in computer system behavior has engendered several different approaches, including new performance measures and new techniques such as distributed scheduler [STAN84]. Efficient use of such systems may be seen as maximizing throughput, as before, but it can also be seen as maximizing the average processor utilization, maximizing the minimum processor utilization or minimizing the communication time. W. W. Chu and others [CHUL84a, CHUL84b] have presented models of distributed processing systems which focus on the communications between processes and the delays these cause in order to provide methods of prediction and performance analysis more relevant to these systems. Y.-C. Chow [CHOW79] and T. C. K. Chou [CHOU82] have studied the question of load balancing in these systems as a dynamic problem. Load balancing methods address the problem of task allocation by attempting to keep all processors approximately equally busy. Although this objective is different

from those discussed, it clearly tends to produce system utilization which also improves performance as measured by other scheduling criteria.

The investigation of loosely coupled systems carried out in Chapter III concerned itself entirely with the scheduling problem of minimizing the makespan of precedence related tasks assuming six properties of the communication overhead in the system. These assumptions are listed in Section 3.1 with the intent of imposing sufficient structure on the problem as to be able to treat scheduling with communication overhead as a well-behaved, deterministic activity. If, for example, the time, $d(P, P')$, required to send a single message unit from P to P' were to vary with time due to events outside the control of the scheduler, then it would not be possible to predict actual time lost due to the communications. If communication protocols allowed collisions, once again it would not be possible to predict the communication costs exactly. Without these assumptions, however, it would be possible to carry out non-deterministic analysis given the probabilistic information on collisions and channel speeds.

These six assumptions alone, however, are not enough to obtain reasonable results or scheduling algorithms. Subsequently, Section 3.2 introduced three additional hypotheses: Short Communication Delays, Identical Links, and Fully Connected architecture. It is possible to define a number of deterministic scheduling problems which do not satisfy one or more of these three conditions, thus it is in this area that the author believes that productive research can be done. Section 3.3 commented that the JLP algorithm is no longer optimal if Short Communication Delays does not hold, but no alternative is suggested. J.-J. Hwang, however, does present in his dissertation [HWAN87] a heuristic scheduling strategy, Earliest Task First (ETF), together with the worst-case bound presented as Proposition 3.1 above, which looks promising even with arbitrary communication delays. ETF, for example, produces optimal schedules in the case of a forest of in-trees and with the Sufficient Processors

hypothesis, as does Join Latest Predecessor (JLP), but also does optimally in the case of a forest of out-trees with extremely long communication delays, where JLP fails miserably. Another possibility is to use the JLP/D algorithm of Section 3.3 as a heuristic in case that running duplicate copies of tasks is allowed. It appears that a reasonable worst-case bound is obtainable for this heuristic in the face of arbitrary communication delays.

The Identical Links assumption, which makes all the message transmission rates equal, is probably not such a critical requirement. Removing this assumption may be about the same level of complication as moving from identical processors to homogeneous processors: processors which differ only in speed. Many optimal algorithms have been obtained for scheduling problems in such an environment, although other formerly easy problems become NP-hard [LAWL82, LAGE81b].

The Fully Connected assumption is, perhaps, the least realistic of the restrictions, particularly if many processors are involved. Unfortunately, the alternative—not fully connected—is not one, but a panoply of different problems. Fully Connected not only hypothesizes that it is possible to get from any processor to any other, but also that such communication is direct and contention free. In a single-bus system, communication is direct but contention ridden; in a hypercube system, communication is frequently indirect and experiences queuing delays at intermediate nodes. With more general network topologies, routing becomes a major issue: certain links may suffer high contention and others be essentially contention free. This is an important area of research, since it is here that contact is made with real systems, but each case will have to be approached separately using heuristic algorithms and non-deterministic analysis.

4.2. SIMD and Specialized Architecture Problems

Throughout this work, the underlying model has been that of the general-purpose MIMD computer in which the tasks are thought of as program modules and each processor works asynchronously and independently from every other. Many of the important issues of efficient use of computing power today deal, on the contrary, with vector processors, systolic arrays, dataflow architectures, and other combinations of processors and control systems for which quite different models are needed. The entire discussion in Chapter II applies primarily to the concern for user satisfaction in an interactive environment or other situation in which the turnaround times of sizable jobs are of interest. The results of Chapter III, on the other hand, can also be significant in case the tasks are single instructions or single operations and the DAG models the evaluation of a single expression. The following paragraphs briefly discuss some of these alternative architectures and their scheduling problems.

The vector processor is an example of a synchronous SIMD architecture capable of performing a particular operation simultaneously on some number of different values or pairs of values. As the name suggests, it is ideal for carrying out such vector operations as vector sums and inner products which are typical of many important scientific applications. While the problem of developing efficient algorithms to utilize this specialized architecture is an important area of current research, from the point of view of scheduling, this problem is actually no different from the classical scheduling problems. Once the algorithm is fixed, the whole vector operation is best treated as a single operation, reducing the problem to the equivalent of a single processor problem.

Another closely allied synchronous architecture is that of the systolic array. In this case, however, the processors are typically specialized operators and at least part of the input data to one processor is the output from a neighboring processor. One

or more data streams march through a prescribed sequence of processors in lock-step, the output being taken from one or more of the processors last visited in the sequence. Very similar to this is the dataflow computer, except that in this case the processors are asynchronous, their operations being triggered by the appearance of input data from a neighboring processor. In both cases there are typically a very large numbers of processors with very sparse interconnections; say a rectangular array with each processor connected to the four nearest neighbors. Once again, major research questions for such architectures are methodologies for creating algorithms and designs which are compatible: what is known as "mapping" applications to architectures. Nevertheless, much of the discussion in Chapter III is relevant to dataflow architectures, where tasks may well be considered UET and communication delays short. The sparse interconnections create new considerations, but the communications only go to nearest neighbors and are, therefore, contention free. Optimal scheduling of a DAG representing the precedence relations among instruction-size tasks under these conditions is a challenging area for continued investigation.

4.3. Open Questions

There remain, as must be the case in an actively expanding area of research such as this, many questions whose answers appear close at hand, but which may still be very difficult. Presented below are only the most immediate extensions of this research which the author feels should be attacked next.

- (1) Continue the simulation studies on the same scheduling methods of Chapter II as well as using the definition of $Size(J)$ of a job J (Section 2.5.) to define alternative methods.

- (2) Obtain a tighter bound on the amount by which $Size(J)$ can underestimate the optimal makespan of J . The author believes that $makespan/Size(J)$ is asymptotic to $3/2$ as m becomes large, rather than to 2 as implied by the bound of Proposition 2.2.
- (3) Implement some of the algorithms on a real MPS to study their actual performance. The overhead associated with running the scheduling algorithms is neglected in the theoretical discussion, other than to establish their time complexity. For a dynamic scheduler, this overhead could determine if it is of practical value.
- (4) Determine a worst case bound for the JLP/D algorithm when applied to problems with arbitrary communication delays.
- (5) Prove the conjecture that EJLP is optimal for $m < 6$ and determine if a simple modification makes it optimal for arbitrary m .
- (6) Find a reasonable scheduling policy for a single-bus system. Such a policy must either assume knowledge of the priorities given by the bus protocol or else give only a heuristic method, since bus contention will cause significant delays in message passing.

BIBLIOGRAPHY

- [ADAM74] Adam, T. L., Chandy, K. M., and Dickson, J. R. A comparison of list schedules for parallel processing systems. *Comm. ACM* 17, 12 (Dec 1974), pp. 685-690.
- [AGRA84] Agrawala, Ashok K., Coffman, Edward G., Jr., Garey, Michael R., and Tripathi, Satish K. A stochastic optimization algorithm minimizing expected flow times on uniform processors. *IEEE Trans. on Computers C-33*, 4 (Apr 1984), pp. 351-356.
- [ANGE86] Anger, Frank D., Hwang, Jing-Jang, and Chow, Yuan-Chieh. An $O(n)$ multiprocessor scheduling algorithm for systems with non-negligible communication times. *Technical Report 86-2*, Dept. of Computer and Information Sciences, University of Florida, Gainesville, 1986.
- [BAKE74a] Baker, Kenneth R. *Introduction to Sequencing and Scheduling*. John Wiley and Sons, New York, 1974.
- [BAKE74b] Baker, Kenneth R., and Su, Zaw-Sing. Sequencing with due-dates and early start times to minimize tardiness. *Naval Research Logistics Quarterly* 21 (1974), pp. 171-176.
- [BERR84] Berry, William L., Penlesky, Richard L., and Vollmann, Thomas E. Critical ratio scheduling: Dynamic due-date procedures under demand uncertainty. *IIE Trans.* 16, 1 (Mar 1984), pp. 81-89.
- [BLAZ83] Blazewicz, J., Lenstra, J. K., and Rinnooy Kan, A. H. G. Scheduling subject to resource constraints: Classification and complexity. *Discrete Appl. Math.* 5 (1983), pp. 11-24.
- [BOWE77] Bowen, Bruce D., and Weisberg, Herbert F. *An Introduction to Data Analysis*. W. H. Freeman and Co., San Francisco, 1977.
- [BRAT75] Bratley, Paul, Florian, Michael, and Robillard, Pierre. Scheduling with earliest start and due date constraints on multiple machines. *Naval Research Logistics Quarterly* 22 (Dec 1975) pp. 165-173.
- [BRUN82] Bruno, J. L. Deterministic and stochastic scheduling problems with treelike precedence constraints. In *Deterministic and Stochastic Scheduling*, M. A. H. Dempster, J. K. Lenstra, and A. H. G. Rinnooy Kan, eds., D. Reidel Publ. Co., Dordrecht, Holland, 1982, pp. 367-374.

- [BRUN81] Bruno, J., Downey, P. and Frederickson, G. N. Sequencing tasks with exponential service times to minimize the expected flow time or makespan. *Journal of ACM* 28, 1 (Jan 1981), pp. 100-113.
- [CHEN75] Chen, N. F., and Liu, C. L. On a class of scheduling algorithms for multiprocessors computing systems. In *Parallel Processing*, Vol. 24 of *Lecture Notes in Computer Science*, G. Goos and J. Hartmanns, eds., Springer-Verlag, New York, 1975, pp. 1-35.
- [CHOU82] Chou, T. C. K., and Abraham, J. A. Load balancing in distributed systems. *IEEE Trans. Software Eng. SE-8* (Jul 1982), pp. 401-412.
- [CHOW79] Chow, Yuan-Chieh, and Kohler, W. H. Models for dynamic load balancing in a heterogeneous multiprocessor systems. *IEEE Trans. Computers C-28*, 5 (May 1979), pp. 354-361.
- [CHUL84a] Chu, Wesley W., Lan, Min-Tsung, and Hellerstein, Joseph. Estimation of intermodule communication (IMC) and its application in distributed processing systems. *IEEE Trans. on Computers C-33*, 8 (Aug 1984), pp. 691-699.
- [CHUL84b] Chu, Wesley W., and Leung, Kin K. Task response time model and its applications for real-time distributed processing systems. *IEEE 1984 Proceedings of the Real-Time Systems Symposium* (1984).
- [COFF78] Coffman, E. G., Garey, M. R., and Johnson, D. S. An application of bin-packing to multiprocessor scheduling. *SIAM Journal on Computing* 7, 1 (Feb 1978), pp.1-17.
- [COFF72] Coffman, E. G. Jr., and Graham, R. L. Optimal scheduling for two-processor systems. *Acta Informatica* 1 (1972), pp. 200-213.
- [CONW67] Conway, Richard W., Maxwell, William L., and Miller, Louis W. *Theory of Scheduling*. Addison-Wesley, Reading, Mass., 1967.
- [DEIT84] Deitel, H. M. *An Introduction to Operating Systems*. Addison-Wesley, Reading, Mass., 1984.
- [DEKE83] Dekel, Eliezer, and Sahni, Sartaj. Parallel scheduling algorithms. *Operations Research* 30, 1 (Jan 1983), pp. 24-49.
- [DEOG83] Deogun, J. S. On scheduling with ready times to minimize mean flow time. *Computer Journal* 26, 4 (1983), pp. 320-328.
- [DOLE85] Dolev, Danny, and Warmuth, Manfred. Scheduling flat graphs. *SIAM Journal on Computing* 14, 3 (Aug 1985), pp. 638-657.
- [ELLI86] Ellis, Michael G. Statistical analysis of average turnaround times of a job scheduling simulator for a multiprocessor environment through the use of hypothesis testing. Unpublished senior project, Dept. of Computer and Information Sciences, Univ. of Florida, Gainesville, 1986.

- [FERN73] Fernandez, E. B., and Russell, B. Bounds on the number of processors and time for multiprocessor optimal schedules. *IEEE Trans. on Computers C-22*, 8 (Aug 1973), pp. 745-751.
- [FLYN66] Flynn, Michael J. Very high speed computing systems. *Proc. of IEEE* 54, 12 (Dec 1966), pp. 1901-1909.
- [FUJI69] Fujii, M., Kasami, T., and Ninomiya, N. Optimal sequencing of two equivalent processors. *SIAM Journal Appl. Math.* 17 (1969), pp 784-789.
- [FUJI71] Fujii, M., Kasami, T., and Ninomiya, N. Erratum. *SIAM Journal Appl. Math.* 20, (1971), p. 141.
- [GABO82] Gabow, Harold N. An almost-linear algorithm for two-processor scheduling. *Journal of ACM* 29, 3 (Jul 1982), pp. 766-780.
- [GARE83] Garey, M. R., Johnson, D. S., Tarjan, R. E., and Yannakakis, M. Scheduling opposing forests. *SIAM Journal Alg. Discrete Methods* 4 (1983), pp. 72-93.
- [GONZ80] Gonzalez, T. F., and Johnson, D. B. A new algorithm for preemptive scheduling of trees. *Journal of ACM* 27, 2 (Apr 1980), pp. 287-312.
- [GRAH69] Graham, R. L. Bounds on multiprocessing timing anomalies. *SIAM Journal of Appl. Math.* 17, 2 (Mar 1969), pp. 416-429.
- [GRAH79] Graham, R. L., Lawler, E. L., Lenstra, J. K., and Rinnooy Kan, A. H. G. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Ann. Discrete Math.* 5 (1979), pp. 287-326.
- [HORN74] Horn, W. A. Some simple scheduling algorithms. *Naval Research Logistics Quarterly* 21 (1974), pp. 177-185.
- [HORO74] Horowitz, Ellis, and Sahni, Sartaj. Computing partitions with applications to the knapsack problem. *Journal of ACM* 21, 2 (Apr 1974), pp. 277-292.
- [HU 61] Hu, T. C. Parallel sequencing and assembly line problems. *Opns. Res.* 9, 6 (Nov 1961), pp. 841-848.
- [HWAN86] Hwang, J.-J., Anger, F. D., and Chow, Y.-C. Problems on multiprocessor scheduling arising from interprocessor communication overhead. *Technical Report 86-1*, Dept. of Computer and Information Sciences, University of Florida, Gainesville, 1986.
- [HWAN87] Hwang, J.-J. "Deterministic Scheduling in Systems with Interprocessor Communication Time." Ph. D. Dissertation, Computer and Information Sciences Department, University of Florida, 1987.

- [KASA84] Kasahara, Hironori, and Narita, Seinosuke. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Trans. on Computers C-33*, 11 (Nov 1984), pp. 1023-1029.
- [KASA85] Kasahara, Hironori, and Narita, Seinosuke. Parallel processing on robot-arm control computation on a multiprocessor system. *IEEE Journal on Robotics and Automation RA-1*, 2 (Jun 1985), pp. 104-113.
- [KOH75] Kohler, W. H. A preliminary evaluation of the critical path method for scheduling tasks on multiprocessor systems. *IEEE Trans. on Computers C-24* (Dec 1975), pp. 1235-1238.
- [KRUC78] Kruck, David J. *The Structure of Computers and Computations*, Vol. 1. John Wiley and Sons, New York, 1978.
- [LAGE81a] Lageweg, B. J., Lawler, E. L., Lenstra, L. K., and Rinnooy Kan, A. H. G. Computer Aided Complexity Classification of Combinatorial Problems. *Technical Report BW 197-81*, Sichtung Mathematisch Centrum, Amsterdam, 1981.
- [LAGE81b] Lageweg, B. J., Lawler, E. L., Lenstra, L. K., and Rinnooy Kan, A. H. G. Computer Aided Complexity Classification of Deterministic Scheduling Problems. *Technical Report BW 198-81*, Sichtung Mathematisch Centrum, Amsterdam, 1981.
- [LAMS77] Lam, Shui, and Sethi, Ravi. Worst case analysis of two scheduling algorithms. *SIAM Journal on Computing* 6, 3 (Sept 1977), pp. 518-536.
- [LAWL64] Lawler, E. L. On scheduling problems with deferral costs. *Management Sci.* 11 (1964), pp. 280-288.
- [LAWL82] Lawler, E. L., Lenstra, J. K., and Rinnooy Kan, A. H. G. Recent developments in deterministic sequencing and scheduling: A survey. In *Deterministic and Stochastic Scheduling*, M. A. H. Dempster, et. al., eds., D. Reidel Publ., Dordrecht, Holland, 1982, pp. 367-374.
- [LLOY82] Lloyd, Errol L. Critical path scheduling with resource and processor constraints. *Journal of ACM* 29, 3 (Jul 1982), pp. 781-811.
- [MALE82] Ma, P. Y. R., Lee, E. Y. S., and Tsuchiya, M. A task allocation model for distributed computing systems. *IEEE Trans. on Computers C-31* (Jan 1982), pp. 41-47.
- [MART77] Martin-Vega, Louis A., and Ratliff, H. Donald. Scheduling rules for parallel processors. *AIIE Trans.* 9, 4 (Dec 1977), pp. 330-337.
- [McDO86] McDowell, Charles E., and Appelbe, William F. Processor scheduling for linearly connected parallel processors. *IEEE Trans on Computers C-35*, 7 (Jul 1986), pp. 632-638.

- [MORI83] Morihara, I., Ibaraki, T., and Hasegawa, T. Bin packing and multiprocessor scheduling problems with side constraint on job types. *Discrete Appl. Math.* 6 (1983), pp. 173-191.
- [PAPA87] Papadimitriou, Christos H., and Tsitsiklis, John N. On stochastic scheduling with in-tree precedence constraints. *SIAM Journal on Computing* 16, 1 (Feb 87), pp. 1-6.
- [RAMM72] Rammamoorthy, C. V., Chandy, K. M., and Gonzalez, M. J. Jr. Optimal scheduling strategies in a multiprocessor system. *IEEE Trans. on Computers C-21* (Feb 1972), pp. 137-146.
- [RAYW86a] Rayward-Smith, V. J. The complexity of preemptive scheduling given interprocessor communication delays. *Internal Report SYS-C86-02*, School of Information Systems, University of East Anglia, Norwich, U.K., 1986.
- [RAYW86b] Rayward-Smith, V. J. UET scheduling with unit interprocessor communication delays. *Internal Report SYS-C86-06*, School of Information Systems, University of East Anglia, Norwich, U.K., 1986.
- [RUSS84] Russell, Roberta S., and Taylor, Bernard W., III. An evaluation of scheduling policies in a dual resource constrained assembly shop. *III Trans.* 17, 3 (Sept 1985), pp. 219-231.
- [SAHN76] Sahni, Sartaj K. Algorithms for scheduling independent tasks. *Journal of ACM* 23, 1 (Jan 1976), pp. 116-127.
- [SIEG56] Siegel, Sidney. *Nonparametric Statistics for the Behavioral Sciences*. McGraw-Hill, New York, 1956.
- [STAN84] Stankovic, John A. Simulations of three adaptive, decentralized controlled, job scheduling algorithms. *Computer Networks* 8 (1984), pp 199-217.
- [SUPP86] Suppe, Dennis R. A task and job scheduling simulator for a multiprocessor environment. Unpublished senior project, Dept. of Computer and Information Sciences, University of Florida, Gainesville, 1986.
- [ULLM75] Ullman, Jeffery D. NP-complete scheduling problems. *Journal of Computer and System Sciences* 10 (1975), pp. 384-393.
- [WILS86] Wilson, Borden A. Research analysis of jobs and tasks run on a scheduling simulator for a multiprocessor environment. Unpublished senior project, Dept. of Computer and Information Sciences, University of Florida, Gainesville, 1986.

GLOSSARY

CPT	Critical Path Time	(Measure of job length)
CPU	Central Processing Unit	
CSJF	Concurrent Shortest Job First	(Scheduling algorithm)
DAG	Directed Acyclic Graph	
DM	Degree of Multiprogramming	
EJLP	Extended Join Latest Predecessor	(Scheduling algorithm)
ETF	Earliest Task First	(Scheduling algorithm)
FAT	First Available Task	(Scheduling algorithm)
FCFS	First Come First Served	(Scheduling algorithm)
HLF	Highest Level First	(Scheduling algorithm)
JLP	Join Latest Predecessor	(Scheduling algorithm)
JLP/D	Join Latest Predecessor with Duplications	(Scheduling algorithm)
LJF	Longest Job First	(Scheduling algorithm)
LRTF	Longest Remaining Task First	(Scheduling algorithm)
MIMD	Multiple Instruction Multiple Data	(System type)
MISF	Most Immediate Successors First	(Scheduling algorithm)
MPS	Multiple Processor System	(System type)
NP	Nondeterministic Polynomial	
SCPF	Shortest Critical Path First	(Scheduling algorithm)
SIMD	Single Instruction Multiple Data	(System type)
SISD	Single Instruction Single Data	(System type)
SJF	Shortest Job First	(Scheduling algorithm)
SPT	Shortest Processing Time	(Scheduling algorithm)

SRCPF	Shortest Remaining Critical Path First	(Scheduling algorithm)
SRTF	Shortest Remaining Time First	(Scheduling algorithm)
SSJF	Sequential Shortest Job First	(Scheduling algorithm)
TPT	Total Processing Time	(Measure of job length)
UCT	Unit Communication Times	
UET	Unit Execution Times	

APPENDIX A RESULTS OF FRIEDMAN TWO-WAY ANALYSIS OF RANK VARIANCE

I. Results using the First Available Task (FAT) Task-Level Strategy

<i>DM</i> = 5	<i>m</i> = 3	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		4.0	3.0	5.0	1.0	2.0
LongJobs		5.0	2.0	4.0	1.0	3.0
RandomJobs		4.0	3.0	2.0	5.0	1.0
$\chi_r = 4.533$						

<i>DM</i> = 5	<i>m</i> = 5	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		5.0	2.5	4.0	2.5	1.0
LongJobs		2.0	5.0	1.0	4.0	3.0
RandomJobs		5.0	2.0	4.0	3.0	1.0
$\chi_r = 3.400$						

<i>DM</i> = 5	<i>m</i> = 7	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		3.0	2.0	4.0	5.0	1.0
LongJobs		2.0	1.0	5.0	4.0	3.0
RandomJobs		4.0	1.0	2.0	5.0	3.0
$\chi_r = 7.733$						

<i>DM</i> = 5	<i>m</i> = 9	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		1.0	3.0	4.5	4.5	2.0
LongJobs		1.0	2.0	5.0	4.0	3.0
RandomJobs		1.0	2.0	5.0	4.0	3.0
$\chi_r = 11.133$						

<i>DM</i> = 5	<i>m</i> = 11	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		2.0	1.0	5.0	4.0	3.0
LongJobs		2.0	1.0	5.0	4.0	3.0
RandomJobs		1.0	2.0	5.0	4.0	3.0
$\chi_r = 11.467$						

<i>DM</i> = 5	<i>m</i> = 13	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		1.0	2.0	4.0	5.0	3.0
LongJobs		2.0	1.0	5.0	4.0	3.0
RandomJobs		1.0	2.0	5.0	4.0	3.0
$\chi_r = 10.933$						

$DM = 5$	$m = 15$	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		2.0	1.0	4.5	4.5	3.0
LongJobs		2.0	1.0	4.5	4.5	3.0
RandomJobs		2.0	1.0	4.0	5.0	3.0

$\chi_r = 11.467$

$DM = 10$	$m = 3$	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		5.0	4.0	2.0	3.0	1.0
LongJobs		4.0	1.0	3.0	5.0	2.0
RandomJobs		3.0	1.0	4.0	5.0	2.0

$\chi_r = 6.667$

$DM = 10$	$m = 5$	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		5.0	3.0	2.0	4.0	1.0
LongJobs		5.0	4.0	1.0	3.0	2.0
RandomJobs		3.0	2.0	4.0	5.0	1.0

$\chi_r = 7.200$

$DM = 10$	$m = 7$	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		5.0	2.0	4.0	3.0	1.0
LongJobs		2.0	4.0	5.0	3.0	1.0
RandomJobs		3.0	1.0	5.0	4.0	2.0

$\chi_r = 7.467$

$DM = 10$	$m = 9$	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		4.0	5.0	2.0	3.0	1.0
LongJobs		2.0	5.0	3.0	4.0	1.0
RandomJobs		2.0	5.0	3.0	4.0	1.0

$\chi_r = 10.400$

$DM = 10$	$m = 11$	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		5.0	3.0	2.0	4.0	1.0
LongJobs		1.0	2.0	5.0	3.0	4.0
RandomJobs		2.0	3.0	5.0	4.0	1.0

$\chi_r = 3.200$

$DM = 10$	$m = 13$	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		2.0	3.0	4.0	5.0	1.0
LongJobs		2.0	1.0	5.0	4.0	3.0
RandomJobs		5.0	4.0	1.0	2.0	3.0

$\chi_r = 1.333$

$DM = 10$	$m = 15$	SJF	SCPF	Random		
WideJobs		3.0	2.0	4.0	5.0	1.0
LongJobs		1.0	2.0	5.0	4.0	3.0
RandomJobs		2.0	3.0	5.0	4.0	1.0
$\chi_r = 9.333$						
$DM = 20$	$m = 3$	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		4.0	5.0	3.0	2.0	1.0
LongJobs		3.0	5.0	2.0	4.0	1.0
RandomJobs		2.0	3.0	5.0	4.0	1.0
$\chi_r = 7.200$						
$DM = 20$	$m = 5$	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		4.0	3.0	2.0	5.0	1.0
LongJobs		4.0	5.0	2.0	1.0	3.0
RandomJobs		2.0	3.0	4.0	5.0	1.0
$\chi_r = 3.467$						
$DM = 20$	$m = 7$	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		3.0	5.0	2.0	4.0	1.0
LongJobs		2.0	3.0	4.0	5.0	1.0
RandomJobs		1.0	2.0	4.0	5.0	3.0
$\chi_r = 6.933$						
$DM = 20$	$m = 9$	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		5.0	3.0	2.0	4.0	1.0
LongJobs		1.0	5.0	2.0	4.0	3.0
RandomJobs		2.0	1.0	5.0	3.0	4.0
$\chi_r = 0.800$						
$DM = 20$	$m = 11$	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		5.0	4.0	3.0	2.0	1.0
LongJobs		3.0	5.0	2.0	4.0	1.0
RandomJobs		4.0	5.0	3.0	1.0	2.0
$\chi_r = 8.533$						
$DM = 20$	$m = 13$	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		3.0	5.0	2.0	4.0	1.0
LongJobs		2.0	4.0	5.0	3.0	1.0
RandomJobs		3.0	4.0	5.0	2.0	1.0
$\chi_r = 8.267$						

$DM = 20$	$m = 15$	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		5.0	4.0	2.0	3.0	1.0
LongJobs		5.0	2.0	4.0	3.0	1.0
RandomJobs		3.0	5.0	4.0	1.0	2.0
$\chi_r = 6.667$						

II. Results using the Most Immediate Successors First (MISF) Task-Level Strategy

$DM = 5$	$m = 3$	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		4.0	5.0	2.0	3.0	1.0
LongJobs		4.0	5.0	3.0	2.0	1.0
RandomJobs		3.0	4.0	2.0	5.0	1.0
$\chi_r = 9.333$						

$DM = 5$	$m = 5$	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		3.0	4.0	2.0	5.0	1.0
LongJobs		3.0	2.0	5.0	4.0	1.0
RandomJobs		2.0	3.0	5.0	4.0	1.0
$\chi_r = 8.267$						

$DM = 5$	$m = 7$	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		5.0	2.0	4.0	1.0	3.0
LongJobs		4.0	1.0	2.0	3.0	5.0
RandomJobs		1.0	4.0	2.0	5.0	3.0
$\chi_r = 1.333$						

$DM = 5$	$m = 9$	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		4.0	3.0	2.0	1.0	5.0
LongJobs		3.0	1.0	4.0	2.0	5.0
RandomJobs		4.0	1.5	3.0	1.5	5.0
$\chi_r = 9.667$						

$DM = 5$	$m = 11$	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		2.0	1.0	3.5	3.5	5.0
LongJobs		3.0	1.0	4.0	2.0	5.0
RandomJobs		3.5	2.0	3.5	1.0	5.0
$\chi_r = 9.533$						

$DM = 5$	$m = 13$	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		4.0	2.0	3.0	1.0	5.0
LongJobs		3.5	1.5	3.5	1.5	5.0
RandomJobs		3.5	3.5	2.0	1.0	5.0
$\chi_r = 9.933$						

<i>DM</i> = 5	<i>m</i> = 15	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		4.0	3.0	2.0	1.0	5.0
LongJobs		3.0	1.0	3.0	3.0	5.0
RandomJobs		3.5	2.0	3.5	1.0	5.0

$\chi_r = 8.467$

<i>DM</i> = 10	<i>m</i> = 3	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		2.0	5.0	3.0	4.0	1.0
LongJobs		5.0	1.0	2.0	3.0	4.0
RandomJobs		4.0	3.0	5.0	2.0	1.0

$\chi_r = 1.867$

<i>DM</i> = 10	<i>m</i> = 5	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		3.0	5.0	2.0	4.0	1.0
LongJobs		4.0	5.0	2.0	3.0	1.0
RandomJobs		5.0	1.0	4.0	3.0	2.0

$\chi_r = 5.333$

<i>DM</i> = 10	<i>m</i> = 7	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		5.0	1.0	4.0	3.0	2.0
LongJobs		4.0	3.0	2.0	5.0	1.0
RandomJobs		4.0	2.0	3.0	5.0	1.0

$\chi_r = 8.800$

<i>DM</i> = 10	<i>m</i> = 9	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		5.0	4.0	3.0	2.0	1.0
LongJobs		3.0	4.0	2.0	5.0	1.0
RandomJobs		2.0	5.0	4.0	3.0	1.0

$\chi_r = 7.200$

<i>DM</i> = 10	<i>m</i> = 11	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		5.0	2.0	4.0	3.0	1.0
LongJobs		5.0	1.0	2.0	3.0	4.0
RandomJobs		2.0	3.0	5.0	4.0	1.0

$\chi_r = 4.267$

<i>DM</i> = 10	<i>m</i> = 13	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		5.0	4.0	2.0	3.0	1.0
LongJobs		4.0	1.0	2.0	3.0	5.0
RandomJobs		2.0	5.0	4.0	3.0	1.0

$\chi_r = 1.333$

$DM = 10$	$m = 15$	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		5.0	4.0	3.0	2.0	1.0
LongJobs		4.0	1.0	3.0	2.0	5.0
RandomJobs		5.0	2.5	4.0	2.5	1.0

$\chi_r = 5.133$

$DM = 20$	$m = 3$	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		3.0	2.0	4.0	5.0	1.0
LongJobs		5.0	4.0	1.0	3.0	2.0
RandomJobs		3.0	4.0	2.0	5.0	1.0

$\chi_r = 6.667$

$DM = 20$	$m = 5$	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		4.0	5.0	2.0	3.0	1.0
LongJobs		2.0	4.0	5.0	1.0	3.0
RandomJobs		3.0	2.0	5.0	4.0	1.0

$\chi_r = 4.000$

$DM = 20$	$m = 7$	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		5.0	4.0	1.0	3.0	2.0
LongJobs		3.0	4.0	1.0	5.0	2.0
RandomJobs		4.0	2.0	5.0	3.0	1.0

$\chi_r = 4.533$

$DM = 20$	$m = 9$	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		5.0	3.0	2.0	4.0	1.0
LongJobs		5.0	4.0	3.0	2.0	1.0
RandomJobs		3.0	1.0	4.0	5.0	2.0

$\chi_r = 6.133$

$DM = 20$	$m = 11$	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		2.0	3.0	4.0	5.0	1.0
LongJobs		2.0	5.0	4.0	3.0	1.0
RandomJobs		1.0	3.0	4.0	5.0	2.0

$\chi_r = 9.333$

$DM = 20$	$m = 13$	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		3.0	2.0	4.0	5.0	1.0
LongJobs		5.0	3.0	1.0	4.0	2.0
RandomJobs		5.0	1.0	2.0	4.0	3.0

$\chi_r = 7.200$

<i>DM</i> = 20	<i>m</i> = 15	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		3.0	5.0	2.0	4.0	1.0
LongJobs		4.0	2.0	3.0	5.0	1.0
RandomJobs		1.0	4.0	3.0	5.0	2.0

$\chi_r = 7.467$

<i>DM</i> = 30	<i>m</i> = 3	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		2.0	5.0	4.0	3.0	1.0
LongJobs		4.0	2.0	5.0	3.0	1.0
RandomJobs		3.0	4.0	5.0	2.0	1.0

$\chi_r = 8.800$

<i>DM</i> = 30	<i>m</i> = 5	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		5.0	4.0	2.0	3.0	1.0
LongJobs		5.0	3.0	2.0	4.0	1.0
RandomJobs		2.0	4.0	3.0	5.0	1.0

$\chi_r = 8.267$

<i>DM</i> = 30	<i>m</i> = 7	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		2.0	5.0	4.0	3.0	1.0
LongJobs		4.0	3.0	1.0	5.0	2.0
RandomJobs		3.0	5.0	2.0	4.0	1.0

$\chi_r = 7.200$

<i>DM</i> = 30	<i>m</i> = 9	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		5.0	4.0	3.0	2.0	1.0
LongJobs		3.0	2.0	5.0	4.0	1.0
RandomJobs		3.0	5.0	2.0	4.0	1.0

$\chi_r = 6.133$

<i>DM</i> = 30	<i>m</i> = 11	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		1.0	3.0	5.0	4.0	2.0
LongJobs		4.0	3.0	2.0	5.0	1.0
RandomJobs		3.0	2.0	4.0	5.0	1.0

$\chi_r = 7.467$

<i>DM</i> = 30	<i>m</i> = 13	SJF	SRTF	SCPF	SRCPF	Random
WideJobs		2.0	3.0	5.0	4.0	1.0
LongJobs		3.0	2.0	4.0	5.0	1.0
RandomJobs		5.0	2.0	4.0	3.0	1.0

$\chi_r = 8.800$

APPENDIX B STATISTICAL TEST ASSUMPTIONS AT 90% CONFIDENCE LEVEL

I. RANDOM JOBS

(1)	(2)	(3)	(4)	(5)	(6)
5	504	3	NO	SRCPF < RNDM	0.057
5	504	3	YES	SRCPF < RNDM	0.065
5	504	15	NO	SCPF < SJF	0.022
5	504	15	NO	SCPF < SRTF	0.018
5	504	15	NO	SRCPF < SJF	0.015
5	504	15	NO	SRCPF < SRTF	0.012
20	519	15	YES	SRCPF < SJF	0.094
20	519	15	YES	SRCPF < RNDM	0.012

II. WIDE JOBS

(1)	(2)	(3)	(4)	(5)	(6)
5	504	15	NO	SCPF < RNDM	0.042
5	504	15	NO	SRCPF < RNDM	0.042

III. LONG JOBS

(1)	(2)	(3)	(4)	(5)	(6)
5	504	3	NO	SJF < RNDM	0.089
5	504	9	NO	SCPF < RNDM	0.090
5	504	9	NO	SCPF < SRTF	0.018
5	504	9	NO	SCPF < SJF	0.016
5	504	9	NO	SRCPF < SRTF	0.028
5	504	9	NO	SRCPF < SJF	0.020
5	504	15	NO	SCPF < RNDM	0.080
5	504	15	NO	SCPF < SRTF	0.070
5	504	15	NO	SCPF < SJF	0.010
5	504	15	NO	SRCPF < RNDM	0.080
5	504	15	NO	SRCPF < SRTF	0.070
5	504	15	NO	SRCPF < SJF	0.090
5	529	15	YES	SRCPF < RNDM	0.042
5	529	15	YES	SRTF < RNDM	0.023

Note: (1) Degree of multiprogramming (2) Number of jobs
(3) Number of processors (4) MISF
(5) Statistical test (6) P value

BIOGRAPHICAL SKETCH

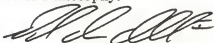
After spending his first seventeen years in Glen Ellyn, Illinois, Frank D. Anger attended Princeton University, graduating Summa Cum Laude in mathematics in 1961. With one year at the University of Hamburg in Germany on a Fulbright Fellowship, he entered the graduate mathematics program in Cornell University, where he obtained his Ph.D. in 1968 with a dissertation in the area of algebraic K-theory. He has subsequently served on the mathematics faculties of M.I.T., University of Kansas, University of Auckland in New Zealand, and, for the past fifteen years, the University of Puerto Rico, where he has been Full Professor since 1982. For the coming year he has accepted an appointment in the Department of Mathematical and Computer Sciences at the Florida Institute of Technology. He is married to Rita Virginia Rodriguez and has three teenage sons.

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



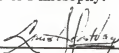
Yuan-Chieh Chow, Chairman
Associate Professor,
Computer and Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



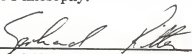
Douglas D. Dankel, II
Assistant Professor,
Computer and Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



Louis A. Martin-Vega
Associate Professor,
Industrial and Systems Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



Gerhard X. Ritter
Professor,
Computer and Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



Stephen M. Thebaut
Assistant Professor,
Computer and Information Sciences

This dissertation was submitted to the Graduate Faculty of the College of Engineering and to the Graduate School and was accepted as partial fulfillment of the requirements for the degree of Doctor of Philosophy.

December 1987

Dean, College of Engineering

Dean, Graduate School